Practical Automatic Loop Specialization

Taewook Oh Hanjun Kim Nick P. Johnson Jae W. Lee† David I. August

Princeton University, Princeton, NJ
{twoh, hanjunk, npjohnso, august}@princeton.edu

†Sungkyunkwan University, Suwon, Korea
jaewlee@skku.edu

Abstract

Program specialization optimizes a program with respect to program invariants, including known, fixed inputs. These invariants can be used to enable optimizations that are otherwise unsound. In many applications, a program input induces predictable patterns of values across loop iterations, yet existing specializers cannot fully capitalize on this opportunity. To address this limitation, we present Invariant-induced Pattern based Loop Specialization (IPLS), the first fully-automatic specialization technique designed for everyday use on real applications. Using dynamic information-flow tracking, IPLS profiles the values of instructions that depend solely on invariants and recognizes repeating patterns across multiple iterations of hot loops. IPLS then specializes these loops, using those patterns to predict values across a large window of loop iterations. This enables aggressive optimization of the loop; conceptually, this optimization reconstructs recurring patterns induced by the input as concrete loops in the specialized binary. IPLS specializes real-world programs that prior techniques fail to specialize without requiring hints from the user. Experiments demonstrate a geomean speedup of 14.1% with a maximum speedup of 138% over the original codes when evaluated on three script interpreters and eleven scripts each.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Code generation, Compilers, Optimization

General Terms Design, Languages, Performance

Keywords Loop specialization, Partial evaluation, Profile based optimization, Program specialization

1. Introduction

Conventional compilers apply optimizations with guarantees of correctness for all valid program inputs [1]. Even if the program input is known at compile-time, the compiler cannot take full advantage of optimization opportunities specific to the program input. Program specialization exploits these opportunities by optimizing a program with respect to a *static input* that is fixed across all invocations of the program. *Static instructions*—those which depend solely on static input or program invariants—always produce the same values across multiple program executions. A compiler generates a specialized program by replacing static instructions with the precomputed values and by residualizing *dynamic instructions*—those which may depend on some non-static inputs. By evaluating static instructions at compile-time, a specialized program generally runs faster than the program generated by the input-unaware compiler.

In many cases, values computed by the static instructions induce repeating patterns across loop iterations. Script interpreters exemplify this property: most script interpreters include an instruction that maintains the program counter of the script. Using this instruction, the interpreter main loop determines the next instruction in the script to dispatch. Since most scripts have loops, a value trace of the program counter exhibits repeating patterns. When these patterns are generated by static instructions, the pattern can be reliably predicted across program runs, since no aspect of the dynamic input will affect those patterns. The interpreter's main loop can be specialized for this pattern by first unrolling the main loop by the length of the pattern and then optimizing each unrolled iteration with respect to the corresponding value.

Many program specializers fail to automatically generate a specialized program that reflects the repetition inherent in the known input. Compile-time analysis to discriminate static and dynamic instructions is too imprecise, thus limiting the specializer's ability to search for repeating patterns among static instructions and in turn limiting application speedup. Existing specializers resort to user annotations to identify those instructions which depend solely on known input [2, 8, 12, 18].

Some specializers [3, 25] do not need user annotations. Instead, they use dynamic information to identify constant or hot values and perform specialization at runtime. However, the cost of dynamic code generation is high. Shankar et al. create multiple specialized instances of the loop for frequently-observed values and apply partial unrolling [25]. This approach introduces dispatch conditions to the hot loop which grow with the number of hot values. To conquer large specialized regions, the number of dispatch conditions grows so large as to limit application speedup.

To overcome these problems, this paper presents Invariant-induced Pattern based Loop Specialization (IPLS). IPLS is the **first fully-automatic program specialization** technique that is applicable to **several complex and widely-deployed script interpreters**. By specializing a program at the granularity of patterns, IPLS enables aggressive optimization across a large window of loop iterations. The choice of patterns over hot values reduces the dispatch overhead.

IPLS is composed of three stages: profiling, pattern detection, and code generation. IPLS profiler identifies static instructions in the program using dynamic information flow tracking and traces the values computed by static instructions. IPLS pattern detector looks for repeating patterns across different iterations in the value trace, which are characteristics of the static instructions. IPLS code generator specializes the program with respect to the detected pattern by unrolling the loop by the length of the pattern and special-

ASPLOS'13, March 16–20, 2013, Houston, Texas, USA.

izing each unrolled iteration to the corresponding value from the pattern.

The performance of IPLS is evaluated using the following set of open-source C programs: the Lua script interpreter [17], the Perl script interpreter [22], and the Python script interpreter [23]. These programs are non-trivial and widely deployed. IPLS automatically specializes these applications and achieves a geomean speedup of 14.1% while increasing program size only by 7.0%. To the best of our knowledge, IPLS is the first program specialization technique that profitably specializes all of these script interpreters.

The primary contributions of this work are:

- The first fully-automatic technique powerful and robust enough to specialize complex applications such as the Perl, Python and Lua interpreters;
- Profiling and analysis to identify fixed input-driven patterns across loop iterations; and,
- Implementation and evaluation on real-world applications.

2. Motivation

Compilers conservatively optimize programs for all valid inputs to guarantee correctness, even though some inputs are effectively invariant. To achieve greater optimization, program specialization allows a compiler to optimize a target program against a specific static input. Here, the *target program* is the program for a specializer to optimize, and *static inputs* are the inputs that are unchanged across program invocations. *Dynamic inputs* are all other inputs; the specializer assumes that dynamic inputs may change across invocations.

2.1 Example: Script Interpretation

Script interpretation is a key application domain for program specialization [14]. A script interpreter executes the same script multiple times with different inputs. Since the script is an input to the interpreter, and it is often reused without a change, the interpreter and the script can be considered as a target program and a static input. The inputs to the script are dynamic inputs. If the script is repeatedly executed, program specialization can optimize the interpreter against the script creating an interpreter which performs better on that script. This accelerates future runs of the script.

Figure 1 illustrates how a script interpreter is specialized for a script. The interpreter has a main loop that reads, parses and executes instructions such as `ADD` and `FOR`. The script has a loop that accumulates values from zero to the dynamic input value. Due to the loop in the script, the main loop of the interpreter iterates over the same sequence of values (i.e. instructions from the script) multiple times. As shown in Figure 1(b), the same code blocks are executed repeatedly to create an input-driven loop.

A program specializer can specialize the interpreter for the input-driven loop of the repeated operations. As Figure 1(c) illustrates, the specializer detects the repeated pattern induced by the static input, and creates a customized loop optimized for the repeating values in `OPC` (i.e. `FOR` and `ADD`). By stitching several iterations into a sequence, many bookkeeping instructions become unnecessary and can be removed by later optimization. The final specialized interpreter is shown in Figure 1(d), and the new specialized program reflects the loop structure within the script.

2.2 Benefits of Pattern based Loop Specialization

Existing fully-automatic program specializers fail to effectively specialize programs with input-driven loops. For example, the specializer proposed by Bala et al. [3] creates only one specialized loop for each hot loop of the program and does not optimize the loop across iteration boundaries. If their specializer is used for the

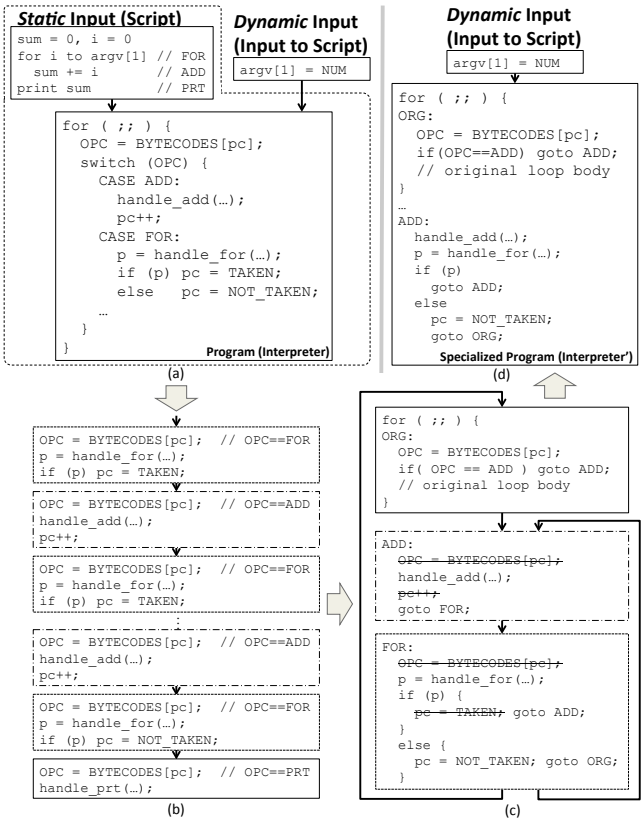


Figure 1. A static input script induces a repeating pattern in variable `OPC`. The interpreter can be specialized with respect to the input script by exploiting this repetition: (a) program, static and dynamic inputs, (b) trace of recurring values across loop iterations, (c) loop iterations stitched into a specialized loop, (d) final specialized code.

example program in Figure 1, it would create a specialized loop for the case of either `OPC==ADD` or `OPC==FOR`, but not both.

Shankar et al. overcome this limitation by creating a specialized loop for each of the hot values observed in the original loop [25]. When applied to the program in Figure 1, they create two specialized loops: for both `OPC==ADD` and `OPC==FOR`. However, this approach suffers from dispatching overhead proportional to the length of the input-driven loop. Although the input-driven loop in the script in Figure 1(c) has a length of only two (with `ADD` and `FOR`), it can be much longer in general. Inserting dispatch instructions for each of the repeating values can incur significant control-flow overhead. A specializer may impose an upper bound on the maximum number of the input-driven loop to limit the control-flow overhead, but only at the cost of lost specialization opportunities. Unlike these approaches, IPLS streamlines the execution of multiple iterations in the original loop associated with a pattern by maintaining one dispatch condition for each specialized pattern.

Another issue with the previous approaches is frequent exits from specialized codes. Figure 2 shows a script with skewed execution paths. The path with {`FOR, IF, SUB`} will be taken more frequently than that with {`FOR, IF, ADD`}. A hot-value based specializer may classify `FOR`, `IF`, and `SUB` as hot, but not `ADD`. Since it does not generate a specialized loop for `OPC==ADD`, it jumps back to the original dispatch routine when the next `OPC` value is `ADD`. In the next iteration it will jump back to the special-

```
sum = 0, i = 0
for i to argv[1]       // FOR
  if rand() % 4 == 1   // IF
    sum += i           // ADD
  else
    sum -= i           // SUB
print sum              // PRT
```

Figure 2. An input script featuring biased control flow.

ized loop for OPC==FOR. As a result, the specializer introduces unnecessary back-and-forth transfers of program control between the specialized loop and the original loop. IPLS avoids this problem by capturing the pattern of {FOR, IF, ADD|SUB} using a loop detection algorithm and generating specialized codes with respect to this pattern. Specializing less-frequently-taken paths in an input-driven loop may increase the code size, but our evaluation shows that increase in binary size is moderate.

3. Overview of IPLS

This section describes IPLS using a simple script interpreter and an input script as an example. Figure 3(c) shows the control flow graph (CFG) representation of the main loop of the interpreter for a simple script language and Figure 3(b) is an input script to that interpreter. The input script is transformed into a sequence of opcodes: FOR, ADD, and PRT. The interpreter's main loop fetches an opcode, branches to the corresponding opcode handler, and repeats. If the input script executes many times, specializing the interpreter with respect to the input script will be highly beneficial.

In Figure 3(c), basic blocks A, B, C, D, ADD, MUL, FOR and PRT compose the main loop while PRE is a preheader block. Individual instructions of the basic blocks are expressed in a medium-level compiler intermediate representation. In the loop header block A, OP loads the next opcode from address ADDR. The next opcode is fetched into OP and the interpreter branches to the basic block corresponding to the opcode: if the value of OP is FOR, the interpreter jumps to the basic block FOR, and so on. The value of ADDR comes either from the loop preheader PRE when the loop is invoked, or from basic block D through the loop backedge.

The workflow of IPLS consists of three stages as depicted in Figure 3(a): profiling, pattern detection and code generation. The rest of this section will provide a high-level overview of these three stages using the example interpreter and script. Figures 3(d) through (f) show the output of each stage while IPLS specializes the interpreter.

3.1 Profiling

The first stage of IPLS is profiling. The profiler instruments the target program and runs the instrumented program with the static input. The values of the static input propagate along the data flow of the program to identify some instructions as static. A static instruction always produces the same value since it depends only on program invariants including the static input, hence can be precomputed at compile-time.

The goal of the profiling stage is to identify static instructions and the values they compute, thus enabling aggressive constant propagation and control flow optimization in the later stages. Towards this goal the profiled executable collects the following information:

- Values computed by the static instructions for each iteration of the loop;
- Address-value pairs for static load instructions (a load instruction is static if the result of the load is computed from a static instruction);
- A set of distinct control-flow paths through each iteration of the loop;
- The number of instructions affected by each static instruction in an iteration; and
- Addresses of all basic blocks within the loop and all functions within the program.

To find static instructions within the program precisely without user annotations and/or heroic static analysis, IPLS employs Dynamic Information Flow Tracking (DIFT) [26]. The only information required from the user is *which* program inputs IPLS may assume fixed across different executions. For example, a user simply directs a script is a fixed input to the interpreter, but inputs to the script are not. This information is propagated along the data flow of the program by instrumented instructions, hence it is possible to decide whether each instruction depends only on static input. Implementation details will be discussed in Section 4.

Figure 3(d) depicts the results of profiling for the example program in Figure 3(c). Since ADDR points to an opcode derived from the static input script throughout execution, both ADDR and OP are classified as static. Therefore, the values of ADDR and OP are traced every iteration. The basic blocks executed in each iteration are also traced as represented by black boxes in Figure 3(d). The values of ADDR and OP constitute an address-value pair of the static load instruction for each iteration. For example, during the first iteration, the address-value pair is <P, ADD> since the value of ADDR equals to P and the value of OP to ADD.

3.2 Pattern Detection

The pattern detection stage interprets profiling results to identify repeating patterns of values computed by static instructions. Such a repeating pattern suggests the existence of an input-driven loop. One can customize this input-driven loop via constant propagation and control flow optimization to generate a specialized loop that efficiently executes those iterations covered by the pattern.

Since there are multiple static instructions within the loop, the patterns they generate may suggest multiple ways to specialize the loop. In Figure 3(d), two static instructions generate two patterns (ADDR = [P, P+1] and OP = [FOR, ADD]). These two patterns are of the same length, and produce the same control flow pattern, though this is generally not the case. The specializer frequently must select among several specialization strategies using heuristics described in Section 5.

The pattern detector's choice of pattern for specialization determines the *dispatching instruction*, i.e. the static instruction whose value will control the path taken during each iteration. While executing the loop, if the dispatching instruction computes the value at the beginning of the pattern, specialized code blocks are dispatched. In the example, ADDR = ϕ(PRE, D) is the dispatching instruction and the pattern is ADDR = [P, P+1]. The dispatch condition compares ADDR to P, and if equal, dispatches into specialized code.

A detected pattern can be represented as a graph called *Meta-Level Loop*. Each value in the pattern, which is produced by the dispatching instruction, corresponds to a node in the graph. The graph has an edge if the values corresponding to the nodes are generated from the adjacent iterations of the loop. By representing the pattern as a graph, it is possible to capture more complex patterns than a repeating sequence of values.

In addition, the pattern detection stage also outputs information about heap constants by analyzing the address-value pairs of static

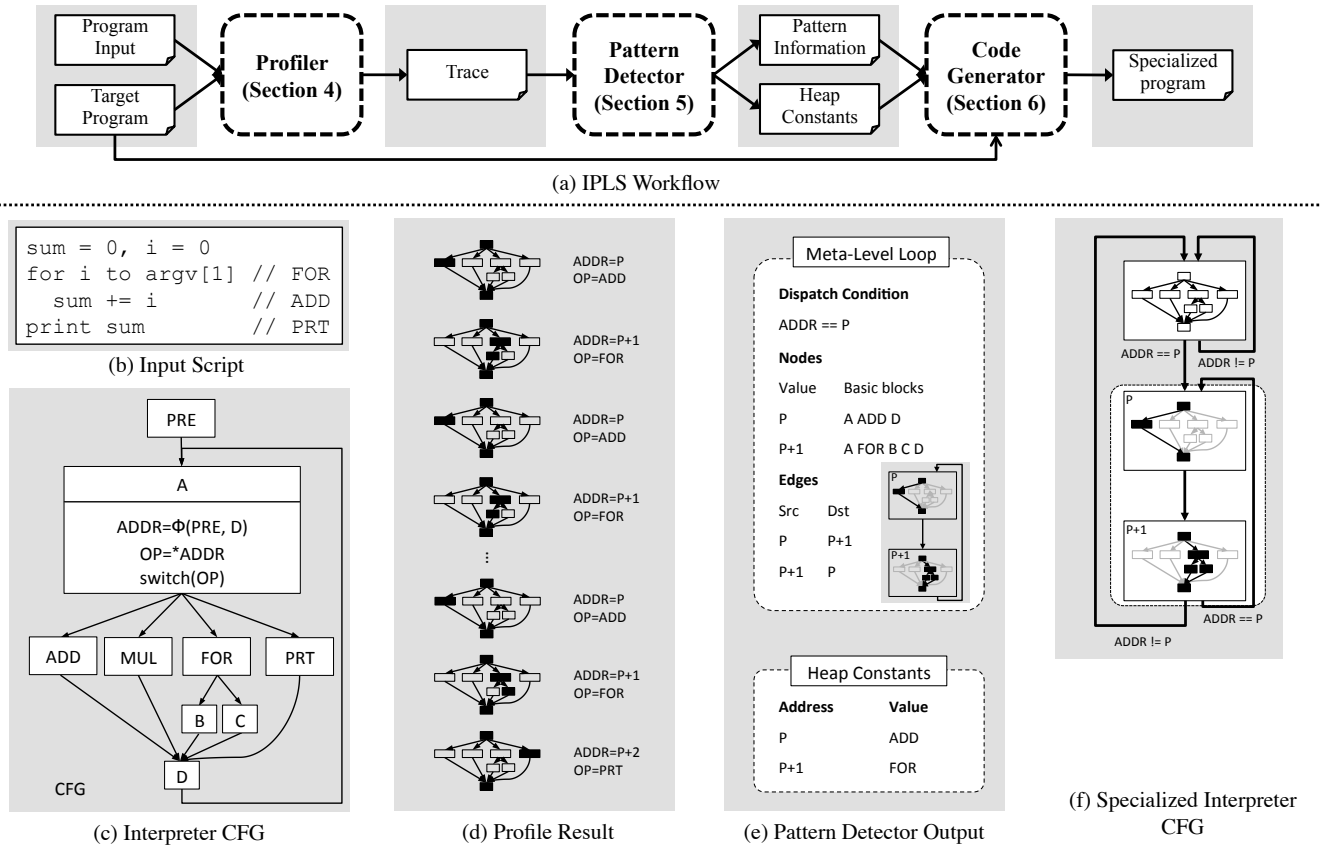


Figure 3. IPLS Specialization: (a) the high-level structure of IPLS, (b) a fixed, static input script, (c) CFG of a script interpreter, (d) result of profiling, including a pattern of static values and their associated iteration control traces, (e) result of pattern detection, and (f) the loop produced by code generation.

load instructions taken in the profiling stage. IPLS assumes that, if the value v corresponding to the address a is identical across the all address-value pairs and the pair of $\langle a, v \rangle$ appeared more than twice, address a holds a constant value v. The code generator specializes a program using this information about heap constants. For example, the static load instruction `OP=*ADDR` in Figure 3(d) always loads `ADD` at address `P` and `FOR` at address `P+1`, so the two heap constants are passed to the code generator.

Figure 3(e) shows the output of the pattern detection stage. A meta-level loop describes the pattern induced by the instruction `ADDR = Φ(PRE, D)`, with two nodes `P, P+1` and two edges `(P, P+1), (P+1, P)`. The code specialized with respect to this meta-level loop will be dispatched if the value of `ADDR` becomes `P`. Each node contains a set of basic blocks which are invoked while tracing the node. If different iterations generating the same value invoke different basic blocks, basic block information stored in the meta-level loop takes a union of them. For example, in Figure 3(d), two iterations generating value `P+1` invoke basic blocks {`A, FOR, B, D`}, while the other invokes {`A, FOR, C, D`}. Therefore, a union of those two is reported as basic blocks corresponding to the value `P+1` in the meta-level loop. The code generator uses this basic block information when creating code blocks specialized for the corresponding value.

3.3 Code Generation

The code generation stage creates specialized codes for each meta-level loop identified in the pattern detection stage. It duplicates the original loop for each node in the meta-level loop and specializes the duplicated loop with respect to the value corresponding to the meta-level node. The code generator also inserts instructions to dispatch the specialized codes.

IPLS specializes a loop by first creating a special version for each iteration (meta-level node). When IPLS duplicates and specializes a loop iteration, it duplicates only those basic blocks listed in the meta-level node. This not only serves to minimize code growth caused by specialization, but also simplifies the control flow of the specialized loop. These simplifications enable more instruction level parallelism in a meta-level node.

The code generator then inserts branch instructions to link multiple specialized iterations into a specialized meta-level loop. In the example, it adds branches from the end of the specialized loop for `ADDR == P` to the head of the specialized loop for `ADDR == P+1` and vice versa, reflecting the two meta-level loop edges `(P, P+1)` and `(P+1, P)`.

These branches are *unconditional* if both of the following conditions are met. First, the next iteration must execute. Second, the value computed by the dispatching instruction of this meta-level loop is equal to the value that the next meta-level loop node is specialized for. For example, a branch from the specialized loop for node `P` to node `P+1` will be an unconditional jump, if it is guaranteed at the end of the specialized loop for node `P` that the loop executes at least one more iteration and that the value of `ADDR` will be `P+1` at the following iteration.

If these conditions cannot be guaranteed at specialization time, the branch must be *conditional*. For example, at the end of the specialized loop for node `P+1`, if the value of `ADDR` for the next instruction is only known at runtime, a conditional branch instruction whose predicate value is `ADDR == P` will be inserted to reflect the meta-level loop backedge from node `P+1` to node `P`.

The code generator also exploits heap constants to perform specialization across load instructions. For example, when specializing the duplicated loop with respect to the value `P` of `ADDR` in Figure 3, no further specialization is possible without the information that address `ADDR` holds a heap constant value `ADD`. Since information about heap constants is acquired by profiling, IPLS inserts instructions to verify the assumed constant value against actually loaded values into the specialized code. This information breaks dependences between the load instruction and its users, hence exposing additional instruction-level parallelism.

Together with address information about basic blocks and functions provided by the profiler, information about heap constants is used to specialize indirect branches and indirect function calls. If a branch/function call target address is derived solely from a heap constant and the address matches the starting address of a basic block or function in the program, an indirect branch/function call can be replaced by a direct branch/function call to the target address. This replaced branch/function call is guarded by a comparison instruction to confirm the heap constant value. This transformation potentially reduces pipeline stalls by simplifying the program's control flow.

Figure 3(f) depicts the structure of the final optimized code after specialization guided by the meta-level CFG and heap constants in Figure 3(e). The main loop of the original interpreter (top white box) dispatches the specialized meta-level loop when the value of `ADDR` is equal to `P`. Specialized codes (round dotted box) are created by duplicating the original main loop twice for each meta-level loop node and specializing each of them using information about meta-level CFG and heap constants. Only necessary basic blocks (colored black) are duplicated during specialization. Under the assumption that the value of `ADDR` of next iteration is guaranteed to be `P+1` at the end of the specialized loop for node `P`, the branch from node `P` to `P+1` is unconditional. However, branch from node `P+1` to `P` is conditional because the value of the `ADDR` for the next iteration cannot be determined statically at the end of the specialized loop for node `P+1`.

4. Profiling

The profiling stage of IPLS collects information to enable the compiler to specialize the program with respect to a given static input. As described in Section 3.1, the IPLS profiler performs a variation of value profiling, load profiling, and path profiling.

Two optimizations distinguish the IPLS profilers from related techniques: (i) the IPLS profiler restricts its observations to static instructions, and (ii) it restricts the scope of value profiling to operations within the header of the target loop. These restrictions limit profiling results to safe specialization assumptions, yet provide information strong enough to support aggressive program specialization. The insight that allows these optimizations is that a specializer may only transform according to *static* program values, and that the dispatch condition (Section 5) must be computable at the start of *every iteration*.

Before profiling, the user marks some portion of the program input as static, such as the input script of an interpreter. We say an instruction is *static* if all of its operands (including values read from memory) are static values. If an instruction is static, then the sequence of values which that instruction generates during program execution is invariant across program executions. This property makes static instructions good candidates for program specialization. The IPLS profiler only collects information pertaining to *static* inputs. By employing DIFT, IPLS collects information related only to the *static* input, obviating the need for heroic static analysis.

```
FILE* fp = fopen(argv[1], 'r');
metadata_fp = metadata_argv[1];
…
fread(buf, 1, size, fp);
metadata_fread = metadata_size & metadata_fp;
memset(metadata_buf, metadata_fread, size*1);
…
```

(a) Instrumented code that reads input

```
OP = *ADDR;
metadata_op = *(metadata_addr);
if (metadata_op == 1)
  printLoadTrace(…);
switch (OP) {
  …
}
```

(b) Instrumented code that uses input

Figure 4. Instrumentation added by the IPLS profiler to achieve dynamic information-flow tracking.

DIFT tags each intermediate value in the program as either *static* or *dynamic*. Profiling instrumentation propagates these tags along the original data flow of the program. To improve DIFT precision, the profiler tracks information flows along register or memory data dependences, but *optimistically* ignores information flows which potentially occur along control dependences. As a consequence, our implementation may report an instruction as static when a conservative implementation would report that values as dynamic (whether or not it actually is dynamic). This improved precision is desirable, since it allows IPLS to perform value profiling on instructions which are *likely* to be static, thus increasing applicability in difficult benchmarks. The trade-off is the risk that specialization will attempt to predict a dynamic value, with no guarantee that the its sequences of values are invariant across program executions.

Incorrectly tagged values have no effect on overall correctness of the specialization procedure, and in practice have minimal effect on IPLS' ability to correctly predict value sequences. There are a few reasons for this. First, IPLS code generation inserts pattern prediction routines at the position of the original hot loop. By virtue of this position, IPLS pattern prediction only executes in cases where the control precondition of the original hot loop is satisfied. This control precondition implies many of the control preconditions of dynamic values in the program, making them effectively static with respect to that position in the program. In the worst case, code generation guarantees safety by falling through to the general loop when all dispatch conditions fail. Mispredictions prevent the general loop from dispatching into the specialized loop, resulting in a lost optimization opportunity (Section 6). Experimental results indicate that sequence prediction is robust and that sequence misprediction has negligible effect on the performance of specialized applications (Section 7).

To support the pattern detector (Section 5), the IPLS profiler performs value profiling on candidate dispatch conditions. The value profiler only observes static instructions *in the loop header*. The dispatch condition must be computable during every iteration of the loop, and all static operations within the loop header satisfy this constraint. This restriction greatly reduces the amount of data

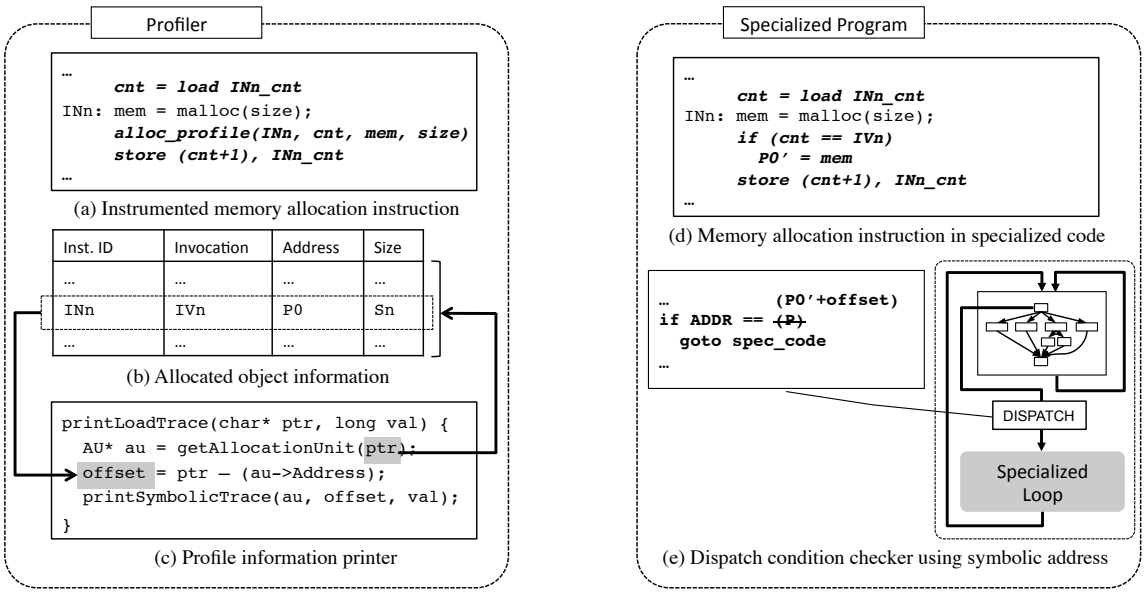


Figure 5. IPLS uses object-relative memory profiling to generate repeatable, symbolic names for relocatable addresses. Variable *INn_cnt* maintains the invocation count of the instruction *INn*.

collected, and in turn reduces the processing overhead of the pattern detector. As a corollary, later analysis of profiling results is insensitive to limited profile coverage, since it ignores operations which do not execute at least once per loop iteration. Note, however, that load profiling is still performed on all static loads within the loop, not only those from the header.

Figure 4 shows the instrumentation to track dynamic information flow. Figure 4(a) is the part of the program which reads the input file specified by the first command line argument. Figure 4(b) is the part which uses the input. `ADDR` in Figure 4(b) points to each element of `buf` in Figure 4(a) through its execution, hence `OP` loads the value read from the input file. Bold and italicized lines are the instructions automatically instrumented by the compiler for tracing metadata and printing profile information.

The code snippet shows that metadata of `argv[1]` is propagated to the metadata of `OP`, through the instrumented instructions. The profiler reports the information related to `OP` only if its metadata set to `1` meaning the value is *static*, which is true only when the metadata of `argv[1]` is `1`. Metadata of command line inputs, which indicates whether the input is static or not, is given by the user.

Since instrumentation is added at an intermediate representation level, instrumentation is not possible of functions whose source code is not available at specialization time. For this reason, tracing the flow of metadata across standard library calls are handled exceptionally via custom information flow tracking instructions. For example, Figure 4(a) shows custom tracking instructions for calls to `fopen` and `fread`.

Another issue of the IPLS profiler is profiling of pointer values. Recall the example in Figure 3 that the dispatch condition of the specialized code was `ADDR == P`. However, the value `P` is a memory address, and there is no guarantee that the absolute pointer value `P` is consistent across across multiple executions of the specialized program. To address this problem, IPLS performs a variation of object-relative memory profiling [27] to derive consistent symbolic names for such pointers.

The IPLS profiler uses symbolic addresses instead of absolute numbers. A symbolic address is a tuple of (instruction ID, invocation counter, offset). *Instruction ID* is an unique static ID of the memory allocation instruction that allocates the object pointed by the address, and *invocation counter* means the number of invocations of the instruction with instruction ID when the object is allocated. *offset* refers to the offset from the base of the object.

Figure 5 describes how the object-relative memory profiling is performed in IPLS. During the execution, the IPLS profiler traces every memory allocation instructions. Figure 5(a) shows the instrumentation to trace memory allocation in bold and italic. For every invocation of memory allocation instruction, `alloc_profile` function call is followed to trace the unique instruction ID of the memory allocation instruction, its invocation counter, start address and size of the allocated object. Traced information is maintained as a table shown in Figure 5(b).

When the profiler outputs the information related to a pointer, it prints the symbolic address tuple instead of absolute number by referring the table. For example, to trace the pointer `P`, which is in range of $[P0, P0 + Sn)$, the profiler finds the corresponding instruction ID and the invocation counter for the object pointed by `P` using the table (`INn` and `IVn` for the example in the figure), calculates the *offset*, and prints those numbers as described in Figure 5(c).

Figure 5(e) shows the use of symbolic address on the specialized program side. Instead of using the absolute value of pointer `P` to check the dispatch condition of the specialized loop, it uses a value `P0′ + offset` which is generated from the symbolic address: `P0′` comes from the `IVn`-th invocation of instruction with ID `INn`, and value `offset` directly comes from the output of the profile. In order to get the value of `P0′` while executing the specialized program, instrumentation is added to the memory allocation instruction `INn`, as shown in Figure 5(d).

5. Pattern Detection

The IPLS pattern detector analyzes profiling information to identify specialization opportunities. The output of this stage includes meta-level loops which represent repeating patterns within the traced values computed by static instructions, and possible heap constants extracted from the trace of static load instructions.

In addition to the meta-level loop, the pattern detector uses the *meta-level trace* to represent the repeating patterns in the program. While the meta-level loop represents patterns across multiple iterations of a single invocation of a loop, the meta-level trace represents patterns across multiple invocations of the loop. Figure 6(a) shows a summarized trace of values generated by a static instruction in a loop across multiple invocations. For the first and second invocations of the loop, the loop iterates for 8 times before it terminates, and for the third invocation it runs for only four iterations. Across first and second invocation of the loop, the static instruction computes exactly the same sequence of values. Therefore, the specializer can exploit patterns which emerge across the *invocations* along with the patterns detected across the *iterations* of a single invocation.

In order to find a meta-level loop or meta-level trace from a given trace of values, the pattern detector first transforms the sequence of trace values into a graph: each traced value becomes a node of the graph, and edges are added between two values adjacent in the trace. Figure 6(b) depicts a graph generated from the trace in Figure 6(a).

To detect a meta-level loop, IPLS runs a natural loop detection algorithm on the graph built from the trace. In the example of Figure 6(b), an edge from node `d` to node `b` forms a backedge because its destination node dominates its source node, and a meta-level loop including nodes `b, c, e`, and `d` can be detected. The dispatch condition for the specialized loop for the meta-level loop is met if the computed value of the static instruction whose profiled values induce the meta-level loop matches the value of the loop header.

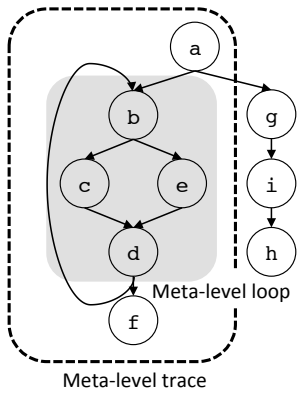
Meta-level traces are created by merging all iteration traces that share the same value for the first iteration. The dispatch condition for the specialized codes for a meta-level trace is determined by the common value from the first iterations. Since the traces of the three invocations shown in Figure 6(a) share the same value `a` generated from the first iteration, Figure 6(b) can be a meta-level trace. However, if the value diverges after the first iteration for different invocations, and some of them appear with a very low probability across invocations, specialized loops for those values will merely increase the program size without much benefit. To prevent this case, only the trace of invocations which share an identical iteration trace are included in the meta-level trace. In Figure 6(a), traces of the first and second invocations are identical, hence are combined in the meta-level trace, yet the trace of the third invocation is excluded because there is no common trace.

At this point the pattern detector may have found several patterns in the trace data. However, the loop can only be specialized according to one pattern. To find the right pattern for specialization, IPLS uses a heuristic based on two measures: (1) coverage of the pattern and (2) number of the instructions affected by the static instruction generating the pattern. IPLS chooses the pattern for which the product of these measures is greatest.

The *coverage* of the pattern is calculated by taking a ratio of the number of iterations covered by the pattern to the total number of iterations in the trace. This measure is related to the benefit of specialization since a higher coverage means that the specialized code likely accelerates a greater portion of the total iterations. The profiler measures the invocation count of the instructions affected by the static instruction *s*. This number is related to the benefit since a higher number implies more computation to be optimized away via precomputation. In the example of Figure 3(d), the pattern of `ADDR = [P, P+1]` and the pattern of `OP = [FOR, ADD]` have the same coverage. While the instruction computing `ADDR` affects two instructions (i.e., `OP=*ADDR` and `switch(OP)`), the instruction `OP=*ADDR` affects only one instruction (`switch(OP)`). Therefore, IPLS chooses to optimize the loop with respect to the pattern of `ADDR = [P, P+1]`. If the coverage of the pattern is less than 0.01, IPLS discards the pattern.

	Invocation		
	0	1	2
Iter 0	a	a	a
Iter 1	b	b	g
Iter 2	c	c	i
Iter 3	d	d	h
Iter 4	b	b	
Iter 5	e	e	
Iter 6	d	d	
Iter 7	f	f	

(a) Example trace



(b) Meta-level loop/trace detected from example trace (a)

Figure 6. Meta-level loops/traces detection extracts a graph which resembles a control-flow graph in which loops are identified.

6. Code Generation

Code generation takes advantage of meta-level loop/trace information and possible heap constant information passed by the pattern detector to specialize codes. Specialized codes are expected to have less computation than the corresponding original codes and be better structured to exploit instruction-level parallelism.

Figure 7 describes how the program represented as a control flow graph in Figure 7(a), which is taken from Figure 3(c), is specialized step by step using the information in Figure 3(e). Throughout this section the code generation process will be explained by walking through the figure.

Step 1: Splitting header and latch block First, IPLS splits the original loop header block and loop latch blocks (blocks which are the source of a loop backedge). As of now, IPLS targets only natural loops for specialization. Natural loops can be canonicalized to have only one backedge and only one latch block. Dispatch instructions for specialized codes are added to the new header block, and the new latch block becomes a point where the control flow merged after the execution of specialized codes. Figure 7(b) shows a new control flow graph after splitting the header and latch blocks.

As shown in the figure, ϕ-nodes placed in the original loop header are moved to new header after splitting. If the dispatch instructions of the specialized codes depend on ϕ-nodes, no other modifications are required. However, if dispatch instructions depend on some other instructions, the instruction and instructions upon which it depends must be cloned into the new header block.

Step 2: Cloning basic blocks To create versions of the loop iteration corresponding to observed traces, the compiler clones basic blocks in the nodes in a meta-level loop/trace. As described in Section 3.3, basic blocks to be cloned for each meta-level node are provided by the pattern detector, which is based on profiling. For a branch instruction in a cloned block, if the original branch target block is also cloned then the instruction is fixed to branch to the corresponding cloned block. If not, the branch jumps to the target block in the original loop. Figure 7(c) depicts the control flow graph after cloning of basic blocks and adjusting branches.

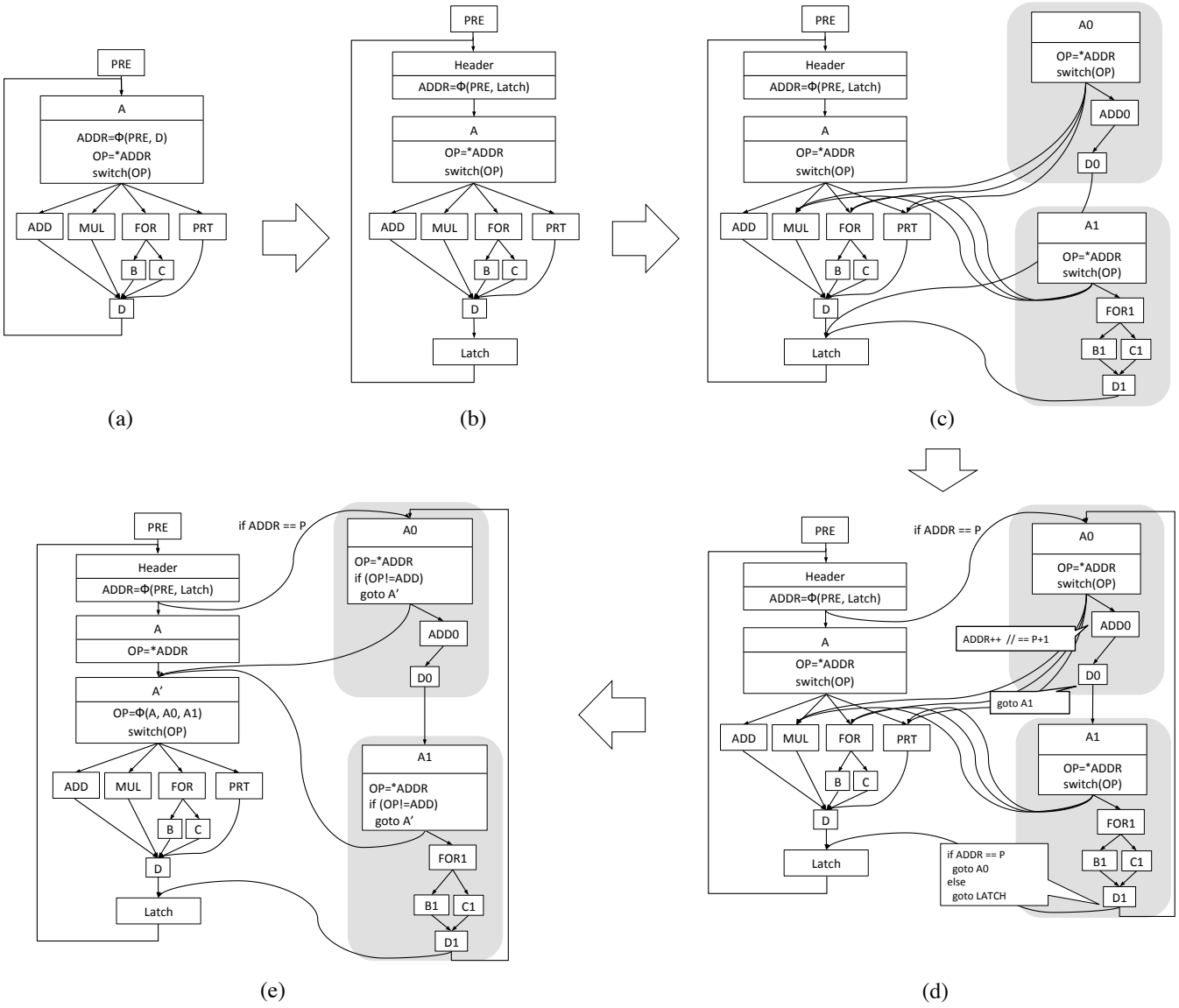


Figure 7. The code generation process: (a) original loop, (b) splitting the loop header and latch, (c) cloning and specializing iterations from a pattern, (d) adding dispatch conditions and stitching specialized iterations into a loop, and (e) adding unexpected exit conditions.

Instead of cloning basic blocks invoked in each node during profiling, one alternative can be analyzing executable basic blocks for each meta-level node and cloning all executable blocks, to minimize the number of returns back to the unspecialized loop. This happens when a branch jumps to a basic block that is never executed during profiling. This approach makes sense because cloned blocks for each meta-level node are specialized for a specific value of the instruction inducing meta-level loop/trace, and specialization may mark a large portion of basic blocks within the loop as unreachable. In practice, the analysis marks every basic block to be executable for non-negligible cases, especially if the specialized loop contains an inner loop. Since cloning the entire loop may lead to code size explosion, IPLS clones the profiled basic blocks only.

Step 3: Adding dispatch instruction and meta-level edges After cloning basic blocks for each meta-level node, dispatch instructions are inserted at the loop header created in Step 1 to invoke the specialized loops. The pattern detector informs the code generator of a dispatch condition for each meta-level loop and trace. Figure 7(d) depicts the control flow edge inserted to dispatch the specialized loop, which is taken when the value of `ADDR` becomes `P` (Actually, the value acquired by the symbolic representation of value `P` is used in dispatch instructions, as described in Section 4, but we use the absolute value `P` here to simplify explanation).

Branches for meta-level edges are also added in this step. The branches are conditioned on the dynamic value of dispatch condition. The branch at the end of block `D1` in Figure 7(d) shows this. The branch checks whether the value of `ADDR` will actually be `P` on the next iteration. If it matches, it jumps to the specialized loop corresponding to the value of `ADDR` being `P`, or returns to the original loop otherwise. If a meta-level node has multiple outgoing edges, IPLS uses a switch statement instead of a branch.

Sometimes a simplified control flow between cloned basic blocks makes branches for meta-level edges unconditional. The branch added at the end of block `D0` in Figure 7(d) is the case. Since blocks `A0` through `D0` have a straightened control flow, and block `ADD0` contains the instruction that increases the value of `ADDR` by one, so the value of `ADDR` for the next iteration is guaranteed to be `P+1` at the end of `D0`. Therefore, without checking, the loop specialized for the condition of `ADDR == P+1` is invoked after the execution of `D0`. Such simplification of the control flow opens opportunities to exploit instruction-level parallelism across different iterations.

Step 4: Exploiting possible heap constants As the last step of code generation, IPLS exploits possible heap constant information.

Figures 7(d) and (e) differ in block `A0`, where the switch instruction has been replaced with a conditional branch. This replacement is possible because the specializer knows (i) `ADDR` must point to `P` and (ii) the pattern detector reports that the memory at `P` holds a heap constant value `ADD`. However, the switch cannot be simply replaced by an unconditional branch. The heap constant information is derived via profiling, which must be verified at runtime; instructions to check the validity of possible heap constant information must remain.

Though it seems that there is no benefit by using heap constant information in block `A0`, performance is improved by replacing switch instructions (often lowered to a jump table in assembly) with conditional branches, which improve the performance of branch prediction. Particularly, the branch prediction is nearly perfect for such cases since the heap constant information is generally true. Although it is not clear in this example, heap constant information also breaks dependences between load instruction and its uses, because the loaded value can be safely assumed to be the expected heap constant after the checking instruction. Breaking dependences creates more optimization opportunities, including better chances for instruction level parallelism.

As an alternative to checking heap constants and branching, speculation assumes all possible heap constants. Also, instead of inserting a comparison between the expected and actual values, it simply logs the comparison result. Then the program occasionally checks the logged value at runtime to see if a *misspeculation* has occurred, and if so, the program *rolls back* to the previous checkpoint of the program, where the program maintains correct state. By removing conditional branches speculative execution opens additional opportunities to exploit instruction level parallelism. However, the overhead of logging and periodic checking of comparison results may negate the benefit coming from more instruction level parallelism. In this paper, we do not use speculative techniques and leave them for future work.

7. Evaluation

A prototype for IPLS is implemented in the LLVM compiler framework [16]. We evaluate it against the following open source C programs: a Lua script interpreter (Lua-5.2.0), a Perl script interpreter (Perl-5.14.2), and a Python script interpreter (Python-2.7.2). All script interpreters first translate the given scripts into an internal intermediate representation and then execute these translated versions. Each interpreter includes a hot loop that performs fetch-execute cycle on the IR. IPLS specialized the fetch-execute loop in each interpreter. All programs are compiled with maximum optimization (`-O3`) using clang compiler version 3.2.

IPLS specializes these C programs against eleven inputs. Each input is a script that is interpreted by the C program. The eleven input programs are selected from the Computer Language Benchmarks Game [7], which are commonly available for all three interpreters and single-threaded (IPLS does not support specialization of multi-threaded programs yet). All evaluations are measured on an Intel Xeon X7460 64-bit processor running at 2.66GHz.

Input	Script (Lines of code)	Iteration coverage(%)	Meta-level-loops/traces
Lua-5.2.0 (19,832 LOC)			
	binary-trees (50)	78.01%	5
	fannkuch-redux (48)	74.75%	6
	fasta (98)	42.63%	5
	k-nucleotide (66)	98.50%	2
	mandelbrot (27)	99.99%	4
	meteor (223)	0.76%	7
	nbody (121)	97.57%	4
	pidigits (104)	99.61%	9
	regex-dna (46)	2.93%	5
	reverse-complement (40)	0.00%	2
	spectral-norm (43)	88.90%	5
Perl-5.14.2 (201,786 LOC)			
	binary-trees (47)	99.99%	4
	fannkuch-redux (55)	99.99%	4
	fasta (122)	98.99%	11
	k-nucleotide (29)	99.93%	2
	mandelbrot (77)	99.90%	4
	meteor (235)	99.90%	10
	nbody (107)	99.90%	10
	pidigits (47)	95.06%	11
	regex-dna (49)	99.92%	3
	reverse-complement (29)	99.99%	4
	spectral-norm (49)	99.99%	4
Python-2.7.2 (314,921 LOC))			
	binary-trees (70)	9.36%	2
	fannkuch-redux (56)	35.02%	3
	fasta (118)	3.03%	4
	k-nucleotide (57)	5.15%	1
	mandelbrot (55)	52.09%	1
	meteor (205)	2.06%	5
	nbody (116)	76.48%	6
	pidigits (40)	1.15%	3
	regex-dna (44)	11.23%	8
	reverse-complement (37)	64.83%	10
	spectral-norm (56)	48.77%	5

Table 1. Execution characteristics of each interpreter and each static input: *Iteration coverage* denotes the fraction of hot loop iterations which are executed in specialized code. *Meta-level loops/-traces* denotes the number of identified patterns for each of which a specialized loop is generated.

Figure 8 shows whole program speedup for the programs specialized with IPLS over original program execution. Figure 9 depicts the program size increase after specialization. As shown, IPLS achieves a geomean speedup of 14.1% in program execution with 7.0% of program size increase.

7.1 Lua-5.2.0

For Lua-5.2.0, speedup of the specialized program correlates to the fraction of iterations executed in the specialized loop. Table 1 shows this trend: *Iteration coverage* column shows the fraction of all iterations of the main loop which execute within specialized code. There is a clear distinction between cases where iteration coverage is less than 3% and cases where iteration coverage is greater than 70%. In the latter case, specialization yields 7.4%–138% speedups, while in the former case, specialization yields a performance degradation.

Low iteration coverage is caused by two factors: unexpected exits due to the limited coverage of path profiling, and value mispredictions which prevent dispatch into the specialized loop.

Input script	Ratio of dynamic instruction counts
binary-trees	1.11
fannkuch-redux	1.13
fasta	1.01
k-nucleotide	1.16
mandelbrot	2.28
meteor	0.99
nbody	1.55
pidigits	1.58
regex-dna	1.00
reverse-complement	1.00
spectral-norm	1.65

Table 2. Ratio of dynamic instruction count of the original program to that of the specialized program for Lua-5.2.0. Larger numbers indicate a greater reduction in dynamic instructions.

Input script	Lua (%)	Perl (%)	Python (%)
binary-trees	19.46	0.00	20.10
fannkuch-redux	2.13	0.00	3.32
fasta	12.50	0.00	0.99
k-nucleotide	0.00	0.00	46.90
mandelbrot	0.00	0.00	1.10
meteor	4.22	0.00	8.67
nbody	0.00	0.00	0.30
pidigits	0.03	0.00	46.83
regex-dna	9.71	0.00	1.66
reverse-complement	0.57	0.00	0.00
spectral-norm	1.25	0.00	8.33

Table 3. Unexpected exits from the specialized loop as a fraction of the number of iterations running in a specialized loop.

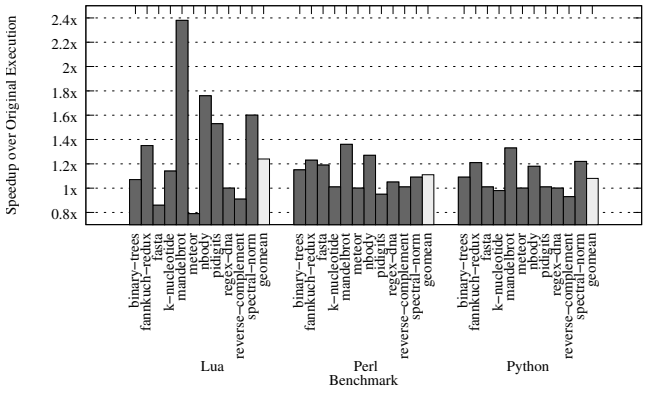


Figure 8. Whole-program speedup with three interpreters: Lua, Perl, and Python, and 11 input scripts for each.

Profile coverage may be limited when a path does not occur during training. Since IPLS generates specialized loops according to path profiling, the program may take an unexpected path within an iteration of its hot loop. To guarantee correctness, the code generator inserts tests which detect this case, and conservatively branches to the unspecialized code. We call such exits *unexpected*. Since the dispatch condition may only enter the specialized loop at the beginning of a pattern, if a specialized loop experiences an unexpected exit the remainder of that pattern must execute in non-specialized code before there is an opportunity to re-enter specialized code. The occurrence of unexpected exits among the total number of iterations is shown in Table 3.

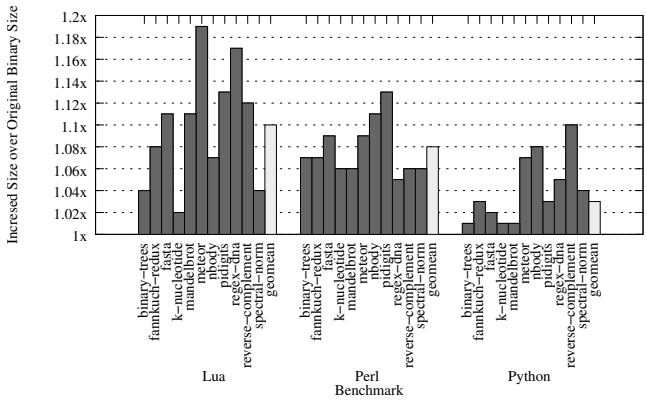


Figure 9. Code size increase after specialization for three interpreters: Lua, Perl, and Python, and 11 input scripts for each.

Additionally, value misprediction may prevent the main loop from dispatching into the specialized loop. This occurs for some input scripts in which control dependences carry information from dynamic input to the dispatch condition. In other words, our optimistic implementation of DIFT occasionally misclassifies a dynamic value as static. As a result, the specialized program may experience a pattern that did not occur during profiling. In such cases, the main loop does not dispatch to the specialized code, decreasing iteration coverage. For example, the hottest loop in `reverse-complement` includes an `if` statement which is predicated on a dynamic input. This induces two different control paths in the script and foils IPLS value prediction.

`Fasta` experiences performance degradation even though the program has good iteration coverage. Difference in dynamic instruction counts after specialization, shown in Table 2, explains this. The numbers in the table show the ratio of dynamic instruction count of the original program to specialized program. Therefore, larger numbers in the table mean that fewer dynamic instructions were executed in the specialized program than the original. Unlike other programs with high iteration coverage, the number is moderate for `Fasta`. This implies that for `Fasta` specialization was not able to find enough precomputable static instructions to amortize the dispatch overhead introduced in the original loop.

7.2 Perl-5.14.2

Table 1 shows that Perl executes more than 99% of its main loop iterations in specialized code for almost all inputs. This suggests that the detected patterns are good predictors of the values. This high predictability stems from the unique implementation of the Perl interpreter's intermediate representation. For instance, `reverse-complement` is implemented using Perl's `split` operation instead of a syntactic `if` statement. Since the Perl interpreter implements large operations such as `split` as a single opcode, the control flow in Perl scripts are typically less dependent on dynamic input.

The Perl interpreter's main loop is structured differently than others. While other interpreters load the opcode, parse it, and branch to the appropriate handler, each handler in the Perl interpreter returns a function pointer that serves as a continuation to the next operation. Hence, the Perl interpreter repeatedly performs indirect calls. This is beneficial for IPLS since there can be no unexpected exits. On the other hand, it limits IPLS since there are fewer opportunities to optimize precomputable instructions.

This suggests that most performance improvements for Perl are caused by increased instruction-level parallelism exposed by un-

rolling the loop, and by better branch prediction caused by replacing indirect function call with conditional branch and direct function call. The ratio of dynamic instructions before and after specialization for Perl ranges from 0.96 to 1.02, except 0.90 in `pidigits`, which shows no benefit from precomputation.

`Pidigits` is the only Perl input which experienced slow down. For `pidigits`, IPLS selects a dispatch condition for 11 specialized meta-level loop/trace. This dispatch condition is generated as 11 load-and-conditional-branch sequences which execute upon every iteration of the main loop. The simple structure of Perl's main loop features such a low fetch-execute overhead that the IPLS dispatch condition is comparably heavy. The benefits of specialization does not overcome the high dispatch overheads and low iteration coverage in the case of `pidigits`.

7.3 Python-2.7.2

Like the Lua interpreter, application speedup for Python generally follows iteration coverage. As shown in Table 1, among the 5 scripts whose iteration coverage is greater than 30%, 4 showed better speedup with Python than other scripts: `fannkuch-redux`, `mandelbrot`, `nbody`, and `spectralnorm`.

Python `reverse-complement`, experiences a slow down despite high iteration coverage. The hottest loop in `reverse-complement` consists of a single meta-level node. Specialization of a singleton meta-level loop has negligible benefit since there are no opportunities for optimization over multiple iterations.

As shown in Table 3, the rate of unexpected exit is generally low, but is significant for three benchmarks: `binary-tree`, `k-nucleotide` and `pidigits`. These unexpected exits have a detrimental effect on iteration coverage and, in turn, on application speedup. Unexpected exits are caused by coverage limitations during path profiling. The Python interpreter's main loop has complicated control flow which is not determined by the current opcode. For example, Python performs reference counting within the main loop and counts the number of operations for preemptive switching among user-threads (although IPLS does not support multi-threaded script, we did not disable the multi-threading features of the interpreter). These rare behaviors are dynamic in the sense that they don't depend on the static input script. Since IPLS uses a natural loop detection algorithm to recognize meta-level loops, not all nodes included in the meta-level loop need to be hot. For the meta-level nodes not frequently observed during profiling, yet which are included in the meta-level loop, their path profile information may lack coverage of complex control flow within the main loop, causing unexpected exits.

8. Related Work

Compile-time Specialization Program specialization requires binding-time information, which classifies all instructions in the target program as either static or dynamic. To obtain the binding-time information, C-Mix [2, 18] and Tempo [8, 9] rely on compile-time analysis and user annotations. Unlike C-Mix and Tempo, IPLS exploits profiling information to obtain the binding-time information, so IPLS is complementary to these previous works.

Berlin et al. [4, 5] propose a specializer that optimizes scientific programs written in high-level languages such as LISP. Since control flows in scientific programs are not affected by input values, they unroll loops to expose parallelism inherent in the underlying numerical computation. However, the loop unrolling may cause a code explosion, so they explicitly exclude loops with high iteration counts from unrolling. Although the authors propose a heuristic to stop unrolling beyond a certain threshold, the heuristic is not evaluated. C-Mix [2, 18] and Tempo [8, 9] also suffer from code explosion, and rely on user annotation to avoid the problem. Since IPLS uses pattern based loop unrolling, this work neither has a code explosion problem, nor requires any user annotation.

JSpec [24] specializes Java using C as an intermediate language and uses Tempo for binding-time analysis. Kleinrubatscher et al. [15] propose a specializer for a subset of FORTRAN using abstract interpretation to gather binding-time information.

Run-time Specialization Run-time specialization [3, 10–13, 19, 25] has an advantage over compile-time specialization because it can exploit run-time constants that are not available at compile-time. However, run-time specialization suffers from high overhead of dynamic code generation. Tempo [10] supports both compile-time and run-time specialization sharing binding-time analysis together, but run-time specialization of Tempo achieves about 80% of the speedup of compile-time specialization, due to the run-time overheads [21]. It shows that specializing the program statically as much as possible can maximize the potential performance of specialized programs. Exploiting profiling results at compile-time, IPLS avoids the run-time overheads.

DyC [12, 13] is a run-time specializer, primarily focused on reducing run-time overheads from dynamic code generation and optimization. DyC requires user annotations to direct optimization policy and improve the precision of binding time information given by compile-time analysis. Although Calpa [20] automatically generates the annotations for DyC with profile information, it is limited to annotations about optimization policy only. Therefore, without programmers hint, Calpa's final result is still limited by compile-time analysis. Unlike DyC, IPLS is a fully-automatic specializer that does not require any user annotation.

Bala et al. [3] and Shankar et al. [25] propose run-time specializers that, like IPLS, do not require any user annotation. The specializers automatically find and optimize frequently executed traces by exploiting the information available at run-time only. Since they detect hot values to find traces, multiple traces in hot code regions can be generated increasing dispatching overheads. However, IPLS detect patterns before specializing codes, so IPLS can reduce dispatching overheads. In addition, while Shankar et al. [25] rely on strong type systems of Java to optimize program with possible heap constants, IPLS can specialize programs written in C language without type system supports.

Bolz et al. and Yermolovich et al. propose Just-In-Time compilers which are optimized to the specific requirements of extracting performance benefits from script interpreters [6, 28]. Script interpreters cannot benefit from tracing-JIT techniques [11, 29] which do not trace across multiple iterations. That is because each iteration in main loops of interpreters has diverse control flow due to different instructions in the script. Addressing the problem, these compilers find frequently executed traces in the scripts that stretch over multiple iterations of the interpreter main loops, and specialize the interpreters for the traces. This approach is similar to IPLS, but the JIT compilers require user annotations to the interpreter program at branch instruction handlers to find boundaries of loops in the scripts, and at data structures to find static values.

9. Conclusion

This paper proposes Invariant-induced Pattern based Loop Specialization (IPLS), the first fully-automatic specialization technique that exploits input-driven patterns across loops. IPLS profiles programs to specialize, and traces how invariants are propagated using information-flow tracking. With the profiling results, IPLS finds repeating patterns across multiple iterations of hot loops, and then specializes the loops predicting values in the loops. Without any user annotation, IPLS specializes three real-world script interpreters and eleven scripts each, yielding a geomean speedup of

14.1% over the original codes yet causing only 7% program size growth.

Acknowledgments

We thank the entire Liberty Research Group for their support and feedback during this work. We also thank the anonymous reviewers for their insightful comments and suggestions. This material is based on work supported by National Science Foundation Grants 0964328 and 1047879, and DARPA contract FA8750-10-2-0253. Jae W. Lee was partly supported by the Korean IT R&D program of MKE/KEIT KI001810041244. All opinions, findings, conclusions, and recommendations expressed throughout this work are those of the authors and do not necessarily reflect the views of the aforementioned funding agencies.

References

[1] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

[2] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 1–12, June 2000.

[4] A. Berlin. Partial evaluation applied to numerical computation. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 139–150, 1990.

[5] A. Berlin and D. Weise. Compiling scientific code using partial evaluation. *IEEE Computer*, 23:25–37, December 1990.

[6] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25, 2009.

[7] Computer Language Benchmarks Game. http://shootout.alioth.debian.org/.

[8] C. Consel, L. Hornof, F. Noël, J. Noyé, and N. Volansche. A uniform approach for compile-time and run-time specialization. In *Selected Papers from the International Seminar on Partial Evaluation*, pages 54–72, 1996.

[9] C. Consel, J. L. Lawall, and A.-F. Le Meur. A tour of Tempo: a program specializer for the C language. *Science of Computer Programming*, 52:341–370, August 2004.

[10] C. Consel and F. Noel. A general approach for run-time specialization and its application to C. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 145–156, January 1996.

[11] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 465–478, 2009.

[12] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. Eggers. Annotation-directed run-time specialization in C. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 163–178, June 1997.

[13] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. Eggers. An evaluation of staged run-time optimizations in DyC. In *Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation*, pages 293–304, May 1999.

[14] N. D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. *ACM SIGPLAN Notices*, 20:82–87, August 1985.

[15] P. Kleinrubatscher, A. Kriegshaber, R. Zöchling, and R. Glück. FORTRAN program specialization. *ACM SIGPLAN Notices*, 30:61–70, April 1995.

[16] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, page 75, 2004.

[17] Lua. http://www.lua.org/.

[18] H. Makholm. Specializing C — an introduction to the principles behind C-Mix/II. Technical report, University of Copenhagen, Department of Computer Science, 1999.

[19] H. Masuhara and A. Yonezawa. Run-time bytecode specialization. In *Proceedings of the Second Symposium on Programs as Data Objects*, pages 138–154, 2001.

[20] M. Mock, M. Berryman, C. Chambers, and S. Eggers. Calpa: A tool for automating dynamic compilation. In *Proceedings of the Second Workshop on Feedback-Directed Optimization*, pages 100–109, November 1999.

[21] F. Noel, L. Hornof, C. Consel, and J. L. Lawall. Automatic, template-based run-time specialization: Implementation and experimental study. In *Proceedings of the 1998 International Conference on Computer Languages*, 1998.

[22] Perl. http://www.perl.org/.

[23] Python. http://www.python.org/.

[24] U. P. Schultz, J. L. Lawall, and C. Consel. Automatic program specialization for Java. *ACM Transactions on Programming Languages and Systems*, 25:452–499, July 2003.

[25] A. Shankar, S. S. Sastry, R. Bodík, and J. E. Smith. Runtime specialization with optimistic heap analysis. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 327–343, 2005.

[26] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 85–96, 2004.

[27] Q. Wu, A. Pyatakov, A. N. Spiridonov, E. Raman, D. W. Clark, and D. I. August. Exposing memory access regularities using object-relative memory profiling. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[28] A. Yermolovich, C. Wimmer, and M. Franz. Optimization of dynamic languages using hierarchical layering of virtual machines. In *Proceedings of the 5th symposium on Dynamic languages*, pages 79–88, 2009.

[29] M. Zaleski, A. D. Brown, and K. Stoodley. YETI: a graduallY extensible trace interpreter. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 83–93, 2007.