Sound and Precise Analysis of Parallel Programs through Schedule Specialization

Jingyue Wu Yang Tang Gang Hu Heming Cui Junfeng Yang
Columbia University
{jingyue,ty,ganghu,heming,junfeng}@cs.columbia.edu

Abstract

Parallel programs are known to be difficult to analyze. A key reason is that they typically have an enormous number of execution interleavings, or *schedules*. Static analysis over all schedules requires over-approximations, resulting in poor precision; dynamic analysis rarely covers more than a tiny fraction of all schedules. We propose an approach called *schedule specialization* to analyze a parallel program over only a small set of schedules for precision, and then enforce these schedules at runtime for soundness of the static analysis results. We build a schedule specialization framework for C/C++ multithreaded programs that use Pthreads. Our framework avoids the need to modify every analysis to be schedule-aware by specializing a program into a simpler program based on a schedule, so that the resultant program can be analyzed with stock analyses for improved precision. Moreover, our framework provides a precise *schedule-aware* def-use analysis on memory locations, enabling us to build three highly precise analyses: an alias analyzer, a data-race detector, and a path slicer. Evaluation on 17 programs, including 2 real-world programs and 15 popular benchmarks, shows that analyses using our framework reduced may-aliases by 61.9%, false race reports by 69%, and path slices by 48.7%; and detected 7 unknown bugs in well-checked programs.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.4.5 [*Operating Systems*]: Threads

General Terms Algorithms, Design, Reliability, Verification

Keywords Specialization, parallel programs, multithreading, control-flow analysis, data-flow analysis, constraint solving

1. Introduction

The computational power provided by multicore hardware and demanded by the massive number of cloud services has made parallel programs increasingly pervasive and critical. Yet, these programs are known to be difficult to get right; they are often plagued with concurrency errors [25], some of which have caused critical failures [23, 30, 37].

A key reason for these bugs is that parallel programs are difficult to analyze using automated tools. These programs typically have an enormous—asymptotically exponential in the total execution lengths—number of execution interleavings, or *schedules*, presumably to react to various timing factors for performance. Static analysis over all these schedules requires over-approximations, resulting in poor precision and many false positives. Dynamic analysis can precisely analyze the schedules that occur, but it rarely covers more than a tiny fraction of all possible schedules, and the next execution may well run into an unchecked schedule with errors.

Our first contribution is a new approach we call *schedule specialization* that combines the soundness of static analysis and the precision of dynamic analysis. Our insight is that not all of the exponentially many schedules are necessary for good performance. A small set often suffices, as illustrated by recent work on efficient deterministic multithreading [16, 17, 24, 28]. Based on this insight, our approach statically analyzes a parallel program over a small set of schedules, then dynamically enforces these schedules. By focusing on only a small set of schedules, we vastly improve the precision of static analysis; by enforcing the analyzed schedules dynamically, we guarantee soundness of the analysis results. Our approach is loosely analogous to previous approaches that combine static analysis and dynamic checking for memory safety (*e.g.*, [26]), but ours aims at parallel programs and schedules.

Schedule specialization may be implemented in many ways for many parallel programming models such as message passing and multithreading; this paper presents one such implementation for C/C++ programs using Pthreads [34]. We represent a schedule as a total order of synchronizations such as lock operations,[1] which can be efficiently enforced [16, 28, 32]. To ensure that schedules are feasible, we collect them from real executions. To enforce schedules, we leverage our PEREGRINE [17] deterministic multithreading system which can enforce a small set of schedules on a wide range of inputs. For instance, it can use about a hundred schedules to cover over 90% of requests in a real HTTP trace for `Apache` [7]. By reusing schedules, we not only make program behaviors repeatable across inputs, but also amortize the static analysis cost in schedule specialization.[2]

A key challenge we face in implementing schedule specialization is how to statically analyze a program w.r.t. a schedule. A naïve method is to make every analysis involved aware of the schedule, but this method would be quite labor-intensive and error-prone. It may also be fragile: if a crucial analysis, such as alias analysis, is unaware of the schedule, it may easily pollute other analyses.

Solving this challenge leads to our second contribution, a program analysis framework with a set of sound and precise algorithms to *specialize* a program according to a schedule. The resultant program has simpler control and data flow than the original program, and can be analyzed with stock analyses, such as constant folding and dead code elimination, for improved precision. In addition, our framework provides a precise def-use analysis that computes *schedule-aware*, *must*-def-use results on memory locations, which can be the foundation of many other powerful analyses (§7).

To specialize a program toward a schedule, our framework works in two steps. It first specializes the control flow by rewriting the program to map each synchronization in the schedule to a

PLDI'12, June 11–16, 2012, Beijing, China.

[1] Users can customize the schedule granularity by selecting which synchronizations go into the schedules (§3).

[2] A set of previously collected schedules may not cover all inputs. If soundness is required on inputs not covered, we may use dynamic analysis (§3).

unique statement. It then specializes the data flow according to the schedule through a series of analyses and rewrites. For instance, if the program loops `N` (a program variable) times to spawn worker threads and the schedule dictates three worker threads, our framework specializes `N`, as well as the other variables where the value of `N` flows to, to be three. The simplified control and data flow can then greatly improve analysis precision.

Our framework has broad applications. For instance, we can build precise static verifiers (*e.g.*, to verify error-freedom) because our verifiers need only verify a program w.r.t. the schedules enforced. Stock compiler optimizations automatically become more effective on a specialized program because it has simpler control and data flow. Our framework can also benefit "read-only" analyses which do not require enforcing schedules at runtime. For instance, we can build precise error detectors that check a program against a set of common schedules to detect errors more likely to occur, while drastically reducing false positive rates. We can perform precise, automated post-mortem analysis of a failure by analyzing only the schedule recorded in a system log to trim down possible causes.

To demonstrate the usefulness of schedule specialization, we have built three powerful applications with high precision, which form our third contribution of this paper. We create a schedule-aware alias analyzer that can distinguish array elements, leveraging the def-use analysis provided by our framework. Moreover, we create a precise static race detector to detect only the data races that may occur when a given schedule is enforced, thus yielding few false positives. Lastly, we create a schedule-aware path slicer [21] to remove irrelevant statements from an execution trace. This slicer can be applied, for instance, to pinpoint the root cause of a failure or generate filters of bad inputs [15].

We have implemented our framework within the LLVM compiler [1] and evaluated it on 17 multithreaded programs, including 2 real programs such as a popular parallel compression utility `PBZip2` [3] and 15 widely used parallel benchmarks from SPLASH2 [4] and PARSEC [2]. Our results show that schedule specialization greatly improves precision: on average, it reduced may aliases by 61.9%, false race reports by 69%, and path slices by 48.7%. The improved precision also helped detect 7 unknown bugs in well-checked [19, 25, 29, 38, 39] programs.

This paper is organized as follows. We first present relevant background on PEREGRINE (§2) and an overview our framework (§3), then present the key algorithms using an example (§4) and detailed descriptions (§5). We then discuss implementation issues (§6), describe the analyses (§7) we build on top of our framework, and show the evaluation results (§8). We finally discuss related work (§9) and conclude (§10).

2. Background: The PEREGRINE System

Our framework leverages PEREGRINE [17] to collect and enforce schedules. This section briefly describes the relevant mechanisms in PEREGRINE; for details of PEREGRINE, please refer to [16, 17].

Collecting schedules. To ensure that schedules are feasible, we collect them from real executions. To do so, we first replace all synchronizations in a program with calls to our synchronization wrappers. At runtime, when a synchronization wrapper is invoked, it appends an entry to a central log file, then calls the underlying synchronization. The synchronization wrappers are properly synchronized so that only one wrapper may run at any time, ensuring that all synchronizations are totally ordered. After the execution finishes, the log contains a schedule. To collect a set of schedules, we repeatedly run a program against representative workloads, and record the schedule in each execution. Although one could use clever heuristics to select schedules, such as iterating through many schedules to pick the fastest one, we currently do not do so.

Enforcing schedules. Given a schedule, we first compute the *preconditions* required for reusing the schedule on later inputs (explained in the next few paragraphs). Then, if an input meeting these preconditions arrives, we enforce the schedule. Specifically, we make all threads call synchronization wrappers following the total order specified in the schedule, and these wrappers are again properly synchronized so that no two are concurrent. Prior results [16, 17, 24, 28] including ours [16, 17] show that enforcing a synchronization order is efficient because most code is not synchronization and can still run in parallel. For instance, this overhead in PEREGRINE ranges from 68.7% faster to 46.6% slower on a diverse set of 18 programs, including the `Apache` [7].

To compute preconditions for a schedule, we first record a detailed execution trace when recording a schedule. We then perform a modified version [17] of *path slicing* [21] on the trace to remove statement instances that do not affect the feasibility of the schedule, *i.e.*, the reachability of all synchronizations in the schedule. Our version of path slicing correctly tracks dependencies (*e.g.*, shared data accesses) *across* threads. Once we compute a path slice for each thread, we *symbolically* execute each slice, tracking the constraints on input data, such as command line arguments and data read from a file or socket. We then use the conjunction of the constraints from each thread as the preconditions of the schedule.

These preconditions have three properties. First, they do not guarantee program termination because we want to sidestep the difficult problem of statically establishing termination. This property is inherited from path slicing [21]. Second, since computing weakest preconditions is undecidable, the preconditions PEREGRINE computes stay on the sound side: if an input satisfies these preconditions, it can be processed by the corresponding schedule (modulo termination). However, these preconditions may preclude inputs that can indeed be processed by the schedule. Third, these preconditions avoid data races, which may cause an execution to diverge, preventing PEREGRINE from enforcing a given schedule. When computing preconditions, PEREGRINE detects potential races that did not occur in a recorded trace, but may occur if we reuse the schedule on different inputs. It computes preconditions sufficient to avoid these races. PEREGRINE makes the races that occurred in the recorded trace deterministic using dynamic instrumentation.

Our PEREGRINE results show that a small number of schedules can cover a wide range of workloads for half of the 18 evaluated programs. We observe that the synchronizations in these programs depend mostly on "meta" properties such as the number of processor cores, and tend to be loosely coupled with the inputs. For these programs, the coverage of a schedule can be extremely high. For instance, consider `PBZip2` which divides an input file evenly among multiple threads to compress. Two schedules are sufficient to compress any file for `PBZip2` as long as the number of worker threads remains the same. (`PBZip2` needs one schedule if the file size can be evenly divided and another otherwise.) Another data point: about a hundred schedules are sufficient to cover over 90% of requests in a real HTTP trace for `Apache` [7]. This *stability* [16] not only makes program behaviors repeatable across inputs, but also reduces the runtime overhead of our framework (§3).

3. Framework Overview

This section presents an overview of our schedule specialization framework by presenting its key definitions and properties.

We assign each statement in a program P a unique static label l_j. An *execution trace*, or a *trace*, of P, denoted by T, is a potentially infinite sequence of dynamic statement instances $i_0, i_1, \ldots$ where each i_j is a tuple $\langle t, l \rangle$, indicating that i_j is an instance of statement l executed by thread t; and if i_j was completed before the start of i_n, then $j < n$ (concurrently executed statements can be ordered either way). Given $i = \langle t, l \rangle$, we use $i.label$ to access l.

A *schedule*, denoted by S, is a sequence of synchronizations $s_0, s_1, \ldots, s_N$ where each s_j is a dynamic statement instance $\langle t, l \rangle$ and l is a synchronization statement. While we anticipate our approach will work well with many parallel programming models, our current framework targets C/C++ programs that use Pthreads. It includes three types of synchronizations in the schedules: (1) Pthread synchronizations such as `pthread_create` and `pthread_mutex_lock` operations; (2) entries and exits of thread functions (functions passed to `pthread_create`) and `main` because these events are natural synchronization points; and (3) additional function calls to resolve ambiguity when a synchronization may be invoked in different calling contexts (explained in §5.1). We call the first two classes *true synchronizations* and the last class *derived synchronizations*. Unless otherwise specified, we use synchronizations to refer to both true and derived synchronizations.

To ensure that schedules are feasible, we collect them from terminating execution traces. A schedule includes only totally ordered synchronizations because our runtime ensures that synchronizations are never concurrent (§2). It need not contain all synchronizations from a trace, enabling flexible tradeoffs between precision and analysis time (see the end of this section).

Different traces may map to the same schedule because traces are more fine-grained than schedules. We use $run(P)$ to denote the set of all traces of program P, including non-terminating ones. These traces cover all possible inputs to P. We use $run_S(P)$ to denote the subset of traces in $run(P)$ with schedule S enforced. Traces in $run_S(P)$ cover only inputs meeting the preconditions of S. Even though S is collected from a terminating trace, another trace with S enforced may still be non-terminating because the preconditions PEREGRINE computes do not guarantee termination (§2). Thus, $run_S(P)$ may include non-terminating traces with a prefix of S enforced.

Without schedule specialization, a static analysis has to consider conceptually all traces in $run(P)$ for soundness. With schedule specialization, a static analysis need consider only traces in $run_S(P)$ for each enforced schedule S, potentially computing much more precise results. One method to use this potential is to modify every analysis to be aware of the schedule and consider a set of executions closer to $run_S(P)$, but this method has the drawbacks discussed in §1.

To avoid modifying every analysis to be aware of schedules, our framework rewrites P into a specialized program P_S for each schedule S so that $run(P_S)$ is close to $run_S(P)$. P_S can then be analyzed with many stock analyses for improved precision. To analyze P over a set of schedules, we can generate a specialized program for each schedule, then merge the analysis results. In addition, our framework provides a constraint-based def-use analysis on P_S that computes precise must-def-use chains on memory locations allowed only by the schedule S. By querying this def-use analysis, stock or slightly modified advanced analyses such as alias analysis can then compute schedule-aware results.

Precision. Ideally for full precision, our framework should make $run(P_S) = run_S(P)$, so that static analysis of P_S considers only traces in $run_S(P)$. In our current framework, $run(P_S)$ may still include traces not in $run_S(P)$ because, without enforcing S with PEREGRINE, an execution of P_S may use a different schedule. Nonetheless, since $run(P_S)$ is much smaller than $run(P)$, stock analyses on P_S should still yield more precise results than on P. Our def-use analysis excludes def-use chains forbidden by the total order in S. That is, it considers traces in $run_S(P_S)$, which our specialization algorithms (§5) guarantee to be $run_S(P)$. The def-use results provided by our framework are thus much more precise than what a stock def-use analysis can compute on P_S.

Soundness. Our framework guarantees that static analysis results on P_S hold for all traces in $run_S(P)$, *i.e.*, whenever schedule S is enforced on program P, because (1) the results from a static analysis on P_S should hold for all traces in $run(P_S)$; and (2) our specialization algorithms guarantee $run(P_S) \supseteq run_S(P)$. In addition, our framework guarantees that the def-use results it provides hold for $run_S(P_S)$, which is $run_S(P)$.

Assumptions and non-assumptions. In our current framework, the soundness of static analysis results is conditioned on that one of the analyzed schedules is enforced. A finite set of schedules collected by PEREGRINE, however, may not cover all inputs. If an input cannot be processed by any of the schedules (or it can but our framework cannot determine this fact), the static analysis results may no longer hold for the execution on this input. If users want to retain soundness on such inputs, they may use dynamic analysis. For instance, if the analysis goal is to verify race freedom, they may use dynamic data race detection on such inputs. If dynamic analysis is used, the overall runtime overhead of our framework may depend on the actual workload, or how frequently it can enforce an analyzed schedule. Fortunately, our PEREGRINE results show that, for many programs, a small set of schedules can cover a wide range of the workloads (§2). For these programs and workloads, our framework rarely needs to use dynamic analysis.

Moreover, even if we cannot use a small set of schedules to cover a wide range of workloads, schedule specialization is still useful for many applications. These applications include all read-only applications such as error detection and post-mortem analysis which do not require soundness on the inputs not covered by analyzed schedules. They also include optimizations because we can simply run the original program on the inputs not covered.

At the algorithmic level, we do not assume data-race-free programs because we intend to keep our framework general and applicable to not just optimizations, but other applications, such as verification and error detection. In particular, data races cannot prevent us from enforcing a schedule (§2). Nonetheless, if optimization is the only goal, we can easily modify our algorithms to exploit this assumption for better results. At the implementation level, our current framework leverages the LLVM compiler, which may indeed assume race freedom. This assumption may be removed by re-implementing the LLVM components we leverage.

Our framework does not require that a schedule collected from a trace includes all synchronizations in the trace. Instead, users can customize the schedule granularity, or which synchronizations go into the schedules, by blacklisting certain synchronization statements. This design enables users to make flexible three-way tradeoffs between precision, analysis time, and schedule reuse-rate. In general, fine-grained schedules have lower reuse-rates and lead to larger specialized programs and longer analysis time, but make the analysis results more precise. One exception is that users should typically blacklist synchronizations hidden behind an abstraction boundary, such as a memory allocator interface, because these synchronizations rarely improve precision but worsen analysis time and reuse-rate. For instance, we blacklisted the lock operations in the custom memory allocator in `cholesky` (§6). An interesting research question is how to optimally make this three-way tradeoff, which we leave for future work.

4. An Example and Algorithm Overview

This section illustrates how our framework operates using an example based on two programs in our evaluation benchmarks: `aget`, a parallel `wget`-like utility; and `fft`, a parallel scientific benchmark.

Figure 1 shows the example program. As can be seen from the code, each thread accesses a disjoint partition of `results`. Suppose we want to build a precise alias analysis to compute this fact, so that we can for example avoid wrongly flagging accesses to `results`

```
1 : int results[MAX];
2 : struct {int first; int last;} ranges[MAX];
3 : int global_id = 0;
4 : int main(int argc, char *argv[]) {
5 :   int i;
6 :   int p = atoi(argv[1]), n = atoi(argv[2]);
7 :   for (i = 0; i < p; ++i) {
8 :     ranges[i].first = n * i / p;
9 :     ranges[i].last = n * (i + 1) / p;
10:   }
11:   for (i = 0; i < p; ++i)
12:     pthread_create(&child[i], 0, worker, 0);
13:   for (i = 0; i < p; ++i)
14:     pthread_join(child[i], 0);
15:   return 0;
16: }
17: void *worker(void *arg) {
18:   pthread_mutex_lock(&global_id_lock);
19:   int my_id = global_id++;
20:   pthread_mutex_unlock(&global_id_lock);
21:   for (int i = ranges[my_id].first; i < ranges[my_id].last; ++i)
22:     results[i] = compute(i);
23:   return 0;
24: }
```

Figure 1: *An example showing how schedule specialization works.* Variable `n` and `p` are two inputs that specify the size of the global array `results` and the number of worker threads, respectively. The code first partitions `results` evenly by the number of worker threads `p` and saves the range of each thread to `ranges` (lines 7–10); it then starts `p` worker threads to process the partitions (lines 11–12). Each worker enters a critical section to set its instance of `my_id` and increment `global_id` (lines 18–20), and then computes and saves the results to its partition (lines 21–22).

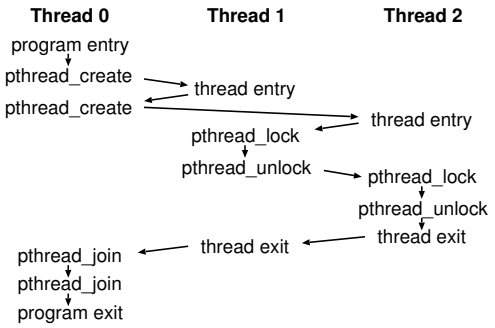


Figure 2: *A possible schedule of the example in Figure 1 when `p` is 2.*

as races for a static race detector. Unfortunately, doing so requires solving a variety of difficult problems. For instance, since `p`, the number of threads, is determined at runtime, static analysis often has to approximate these dynamic threads as one or two abstract thread instances. It may thus collapse distinct accesses to `results` from different threads as one access, computing imprecise alias results. Even if an analysis can (1) distinguish the accesses to `results` by different threads and (2) infer bounds on integers such as array indices using range analysis [33], it may still fail to compute that these accesses are disjoint. The reason is that the array partition each worker accesses depends on its instance of `my_id`, which further depends on the schedule (or the order in which worker threads enter the critical section at lines 18–20). In short, if a static analysis has to consider all possible schedules of this program, it would be very difficult to compute precise results.

Fortunately, these difficult problems are greatly simplified by schedule specialization. Suppose whenever `p` is 2, we always enforce the schedule shown in Figure 2. Since the number of threads is fixed, distinguishing them becomes easy. In addition, since the order in which threads enter the critical section at lines 18–20 is fixed, each worker thread always gets a fixed `my_id` and thus accesses a fixed partition of `results`.

```
1 : ... // same global declarations as in Figure 1
2 : int main(int argc, char *argv[]) {
3 :   int i;
4 :   int p = atoi(argv[1]), n = atoi(argv[2]);
5 :   for (i = 0; i < p; ++i) {
6 :     ranges[i].first = n * i / p;
7 :     ranges[i].last = n * (i + 1) / p;
8 :   }
9 :   i = 0; assume(i < p);
10:   pthread_create(&child[i], 0, worker_CLONE1, 0);
11:   ++i; assume(i < p);
12:   pthread_create(&child[i], 0, worker_CLONE2, 0);
13:   ++i; assume(i >= p);
14:   i = 0; assume(i < p);
15:   pthread_join(child[i], 0);
16:   ++i; assume(i < p);
17:   pthread_join(child[i], 0);
18:   ++i; assume(i >= p);
19:   return 0;
20: }
21: void *worker_CLONE1(void *arg) {
22:   pthread_mutex_lock(&global_id_lock);
23:   int my_id = global_id++;
24:   pthread_mutex_unlock(&global_id_lock);
25:   for (int i = ranges[my_id].first; i < ranges[my_id].last; ++i)
26:     results[i] = compute(i);
27:   return 0;
28: }
29: void *worker_CLONE2(void *arg) {
30:   pthread_mutex_lock(&global_id_lock);
31:   int my_id = global_id++;
32:   pthread_mutex_unlock(&global_id_lock);
33:   for (int i = ranges[my_id].first; i < ranges[my_id].last; ++i)
34:     results[i] = compute(i);
35:   return 0;
36: }
```

Figure 3: *The resultant program after control-flow specialization.* The loops at lines 11–12 and lines 13–14 in Figure 1 are unrolled because they contain synchronizations in the schedule, while the other loops are not. Thread function `worker` is cloned twice, making it easy for an analysis to distinguish the two worker threads. The algorithm adds special `assume` calls to pass constraints on variables to the data-flow specialization step.

To practically take advantage of these observations, our framework specializes a program toward a schedule. It does so in two steps. In the first step, it specializes the control flow of the program by "straightening" the program so that each synchronization in the schedule maps to a unique statement, and the synchronizations within each thread are control-equivalent. Specifically, it first constructs a super control flow graph (CFG) of the program, which includes function call and return edges. It then iterates through each pair of consecutive synchronizations (s_1, s_2) in a thread, and clones the statements between s_1 and s_2 in the CFG. Figure 3 shows the result after specializing control flow. Loops containing synchronizations such as `pthread_create` are unrolled, and thread functions are cloned, making it easy for an analysis to distinguish threads. Note that such cloning and unrolling are selectively done only to functions that may do synchronizations, so they would not explode the size of a specialized program.

In the second step, our framework specializes the data flow through a series of analyses. For instance, it constructs an advanced def-use graph for variables and memory locations according to the schedule, and replaces variables with constant values when it can.

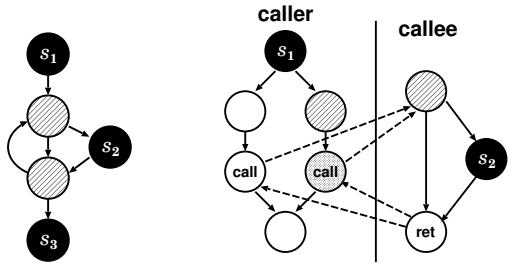
```
... // same global declarations as in Figure 1
int main(int argc, char *argv[]) {
  int p = atoi(argv[1]), n = atoi(argv[2]);
  ranges[0].first = 0;
  ranges[0].last = n / 2;
  ranges[1].first = n / 2;
  ranges[1].last = n;
  pthread_create(&child[0], 0, worker_CLONE1, 0);
  pthread_create(&child[1], 0, worker_CLONE2, 0);
  pthread_join(child[0], 0);
  pthread_join(child[1], 0);
  return 0;
}
void *worker_CLONE1(void *arg) {
  pthread_mutex_lock(&global_id_lock);
  global_id = 1;
  pthread_mutex_unlock(&global_id_lock);
  for (int i = 0; i < ranges[0].last; ++i)
    results[i] = compute(i);
  return 0;
}
void *worker_CLONE2(void *arg) {
  pthread_mutex_lock(&global_id_lock);
  global_id = 2;
  pthread_mutex_unlock(&global_id_lock);
  for (int i = ranges[1].first; i < ranges[1].last; ++i)
    results[i] = compute(i);
  return 0;
}
```

Figure 4: *The resultant program after data-flow specialization.* Variable `p` and `my_id` are replaced with constants, the loop at lines 5–8 in Figure 3 is unrolled, and some dead code is removed.

It does these analyses by collecting constraints from the program and querying a constraint solver. For instance, from lines 9, 11, and 13 in Figure 3, it infers that `p` must be 2, so it replaces `p` with 2. Similarly, it computes a precise def-use chain for the accesses to `global_id`, and infers that each `my_id` instance has a constant value. Once variables are replaced with constants, we can apply techniques such as constant folding, dead code elimination, and loop unrolling to further specialize the program. Moreover, these stock techniques and our analyses can run iteratively, until the program cannot be specialized any more. Figure 4 shows the result of specializing data flow.

Benefits of schedule specialization. The specialized program in Figure 4 has much simpler control and data flow than the original program in Figure 1, enabling stock analyses to compute more precise results. For instance, range analysis can now compute thread-sensitive results because the `worker` function is cloned; it can also compute precise bounds for the loop index i in the `worker` clones because the indice to `ranges` are now constant, not `my_id` which can have a large range if no schedule is enforced.

Our framework guarantees that static analysis results on Figure 4 hold for the set of all traces with the schedule in Figure 2 enforced. This set is fairly large because this schedule can be enforced as long as the number of threads `p` is 2, regardless of the data size `n` or the contents of other input data. A small set of such schedules, including one for each number of threads, can practically cover all inputs because (1) most parallel programs we evaluate achieve peak performance when the number of worker threads is identical or close to the number of processor cores and (2) the number of cores a machine has is typically small (*e.g.*, smaller than 100). This example illustrates the best case of our approach: by analyzing only a small set of schedules, we enjoy both precision and soundness for practically all inputs.



(a) *Intra-procedural ambiguity* (b) *Inter-procedural ambiguity*

Figure 5: *Two types of ambiguity.* The black nodes are synchronizations. The hatched nodes are the statements between s_1 and s_2 computed by the reachability analysis. The gray "call" node is a derived synchronization marked to resolve inter-procedural ambiguity.

5. Algorithms

This section describes our algorithms to specialize the control flow (§5.1) and data flow (§5.2) of a program toward a schedule.

5.1 Specializing Control Flow

To ease the description of our algorithms, we define a *segment*, denoted by G, as the maximal portion of the CFG between two synchronization statements l_1 and l_2 that is free of other synchronization statements. A segment between l_1 and l_2 includes any statement s.t. (1) l_1 can reach this statement in the CFG without reaching any other synchronization statements and (2) this statement can reach l_2 without reaching any other synchronization statements. We denote the set of statements in G by $G.L$, and the set of control-flow edges in G by $G.E$.

To specialize the control flow, we clone the segment between every two consecutive synchronizations within a thread, and then join all cloned segments together to form the specialized program. One difficulty to find all statements between two consecutive synchronizations is *ambiguity*: there may be multiple ways from one synchronization to another. For instance, Figure 5a shows an ambiguous case caused by multiple paths in the CFG. The loop body shown has two paths, only one of which contains synchronization s_2. This loop has to be transformed so that s_1 and s_2 become control-equivalent. Our algorithm to specialize control flow automatically handles this case, described later in this subsection.

Figure 5b shows another ambiguous case caused by multiple paths in the call graph. From s_1 to s_2, we can go through either call. To resolve call-stack ambiguity, we instrument calls to functions that may transitively do synchronizations, and include these *derived synchronizations* in the schedule as well. The result is that we can compute exactly one call stack for each synchronization in a schedule. Our evaluation shows that the number of derived synchronizations is small, and they incur negligible overhead (§8.2).

Algorithm 1 shows how we specialize the control flow of a program. We first explain the helper function SpecializeControlFlowThread, which takes the original program P, a schedule S, and a thread t, and outputs a subprogram P'_t specialized according to t's synchronizations in S. For every two consecutive synchronizations of t, this function does a forward and a backward depth-first search to find the segment between the two synchronizations. It then clones the segment, copying and renaming functions as necessary, and appends the segment clone to P'_t.

Function DFS traverses the CFG from statement l_1 to l_2 and saves the visited portion of the CFG to output parameter G_{out}. It backtracks whenever it meets a synchronization. During the traversal, it maintains the current call stack in cs so that it can compute where to return (l_{ret}). When it reaches l_2, it saves the call stack to output parameter cs_{out}, which SpecializeControlFlowThread will pass to BackwardDFS (in the same iteration) and DFS (in the next iteration). If there are multiple paths from l_1 to l_2, DFS may reach

Algorithm 1: Control-Flow Specialization

Input : the original program P and a schedule S
Output: the specialized program
SpecializeControlFlow(P, S)
  foreach thread t **do**
    $P'_t \leftarrow$ SpecializeControlFlowThread(P, S, t)
  foreach P'_t **do**
    foreach statement l in P'_t **do**
      if l is a `pthread_create` **then**
        $ct \leftarrow$ child thread created by l
        set thread function of l to P'_{ct}
  return $\cup_t P'_t$
SpecializeControlFlowThread(P, S, t)
  $P'_t \leftarrow \emptyset$ // the specialized subprogram for thread t
  $cs \leftarrow \emptyset$ // the call stack of the current synchronization
  $S_t \leftarrow$ the sub-schedule of S for thread t
  for (s_i, s_{i+1}) in S_t **do**
    $G \leftarrow (\{s_i.label\}, \emptyset)$ // segment between s_i & s_{i+1}
    // G and the second cs below are output parameters.
    DFS(cs, $s_i.label$, $s_{i+1}.label$, cs, G)
    BackwardDFS(cs, $s_{i+1}.label$, $s_i.label$, G)
    $P'_t \leftarrow P'_t \cup$ Clone(G)
  return P'_t
DFS(cs, l_1, l_2, cs_{out}, G_{out}) // l_1 and l_2 may be the same
  if l_1 is a derived synchronization **then**
    $l \leftarrow$ entry of l_1's callee function
    TryDFS($cs + l_1$, l_1, l, l_2, cs_{out}, G_{out})
  else if l_1 is a return statement **then**
    $l_{ret} \leftarrow$ the top of cs
    $l \leftarrow l_{ret}$'s intra-procedural successor
    TryDFS($cs - l_{ret}$, l_1, l, l_2, cs_{out}, G_{out})
  else
    foreach intra-procedural successor l of l_1 **do**
      TryDFS(cs, l_1, l, l_2, cs_{out}, G_{out})
TryDFS(cs, l_1, l, l_2, cs_{out}, G_{out})
  $G_{out}.E \leftarrow G_{out}.E \cup \{(l_1, l)\}$
  if $l = l_2$ **then**
    $cs_{out} \leftarrow cs$
  if $l \notin G_{out}.L$ **then**
    $G_{out}.L \leftarrow G_{out}.L \cup \{l\}$
    if l is not a synchronization **then**
      DFS(cs, l, l_2, cs_{out}, G_{out})

l_2 multiple times in one traversal, but each time the call stack must always be identical because call-stack ambiguity is resolved.

The results of DFS may include statements and CFG edges that cannot reach $s_{i+1}.label$. BackwardDFS prunes these statements and edges by traversing G, not the full CFG, backward from $s_{i+1}.label$ to $s_i.label$. It is similar to DFS except it is backward and need not output a call stack, so we omit it from Algorithm 1.

We now explain the main function of this algorithm, SpecializeControlFlow. It first invokes SpecializeControlFlowThread to compute a specialized program based on the synchronizations of each thread. It then replaces the thread function in each `pthread_create` callsite with the cloned thread function. It finally merges all the specialized programs together.

Discussion. We note three subtleties of Algorithm 1. First, it automatically handles the ambiguity in Figure 5a. Suppose the schedule is $s_1 s_2 s_3$. We do not know the loop bound because one path in the loop body calls s_2 and the other does not. However, our algorithm can still specialize the CFG into Figure 6. This feature is critical

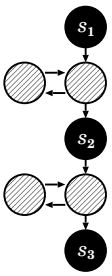


Figure 6: *The specialized CFG of Figure 5a with schedule $s_1 s_2 s_3$.*

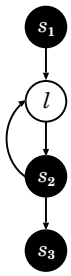


Figure 7: *Example illustrating the need to traverse CFG edges.*

in some cases. For example, if s_2 is `pthread_create`, this feature clones the thread functions, providing thread-sensitivity.

Second, *DFS* traverses into the body of a function only when the call to this function is a derived synchronization, *i.e.*, this function may transitively do synchronization. Thus, the G_{out} it computes includes only CFG portions from such functions, and is not a full segment. The effect is that only these CFG portions are cloned, possibly multiple times, and each clone is *unique* to a segment in the resultant program; other functions are copied verbatim to the resultant program. By doing so, we capture the information from the schedule without exploding the size of a specialized program.

Lastly, to compute a segment, we cannot simply compute only the statements in the segment and copy all edges from the original CFG. To illustrate, consider Figure 7. Suppose the schedule is $s_1 s_2 s_3$. The segment between s_1 and s_2 should include statements s_1, s_2, and l, but exclude the CFG edge from s_2 to l.

5.2 Specializing Data Flow

Given a control-flow-specialized program, we specialize its data flow using a series of constraint-based analyses leveraging the STP [5] integer constraint solver. We collect constraints on two types of program constructs: the LLVM virtual registers and memory locations. LLVM virtual registers are already in static-single-assignment (SSA) form, so collecting constraints on them is relatively straightforward. Table 1 lists how we collect constraints for some LLVM instructions; the remaining instructions are similar and omitted for space. We handle loops by exploiting LLVM's loop structure: most of the loops in our evaluated programs are canonicalized by LLVM so that we can easily collect range constraints on the loop indices.

Our algorithm to collect constraints on memory locations mimics classic def-use analysis. It collects two forms of constraints on memory locations: (1) the value loaded by an LLVM load instruction is the same as the value stored by a store instruction and (2) the value loaded by a load is the same as the value loaded by another load. That is, we treat an LLVM load instruction as a "use" and an LLVM *load or store* instruction as a "def." Treating loads as defs shortens the def-use chains and reduces analysis time.

Collecting precise def-use constraints on memory locations is challenging for multithreaded programs. Fortunately, schedule specialization enables two key refinements over classic def-use analysis so that our algorithm can collect more precise constraints. First, we compute def-use chains spanning across threads because multiple threads may indeed access the same shared memory location. Without schedule specialization, it would be hopeless to track precise inter-thread def-use chains because the defs and uses may pair up in numerous ways depending on the schedules.

Second, we compute *must*-def-use, instead of may-def-use, results. Our insight is that by fixing the schedule, we often fix the def-use chains of key shared data, such as the `global_id` in Figure 1. It is thus most cost-effective to focus on these must-def-use chains because (1) they essentially capture the information (and hence the precision; see results in §8.1) from the schedule and (2)

Instruction type	LLVM instruction	STP constraint	Note
binary	a = b op c	$a = b$ op c	"op" is a binary operator
integer comparison	a = icmp pred, b, c	$a = (b$ pred $c)$	"pred" is an integer comparison predicate such as $<$ and $\neq$
pointer arithmetic	a = gep b, i_1, ..., i_k	$a = b$ + $i_1 \times$ sizeof(*b) + the offset of i_k-th field of i_{k-1}-th field of ... of i_2-th field of b	"gep" calculates the location of a specific field in an array/structure
select instruction	a = select b, v_1, v_2	$b = 0 \rightarrow a = v_1$ $b = 1 \rightarrow a = v_2$	If $b = 0$, $a = v_1$; otherwise $a = v_2$
Φ instruction	$a = \Phi(v_1, \ldots, v_k)$	$a = v_1$ or ... or $a = v_k$	a may have any incoming value
call and return	a = func$(b_1, \ldots, b_k)$ func$(f_1, \ldots, f_k)$ {return r;}	$b_1 = f_1, \ldots, b_k = f_k$ $a = r$	Capture the equalities between the actual and formal parameters/return values

Table 1: Collecting constraints from LLVM instructions on virtual registers.

there are typically much fewer must-def-use constraints than may-def-use constraints, so the constraint-solving cost is low.

To practically implement these refinements, we compute def-use chains only on the instructions unique to each segment. In addition, we look for defs of a use only from its *inter-thread dominators* which always execute before the use once the schedule is enforced. These dominators are dominators on the CFG augmented with edges representing the total order of synchronizations in a schedule. We explain our algorithm as well as these refinements in the next few paragraphs.

Algorithm 2 shows the pseudo code to collect def-use constraints for memory locations. It operates on a control-flow specialized program with each synchronization in the schedule mapped to a unique instruction. It references several symbols defined on the instructions and segments (§5.1) in this program:

- $value(l)$: the value loaded if l is a load instruction, or the value stored if l is a store;
- $loc(l)$: the memory location accessed by instruction l;
- $segment(l)$: the unique (explained in the next paragraph) segment containing instruction l, or $None$;
- $thread(G)$: the thread containing segment G;
- $begin(G)$: the synchronization at the beginning of segment G;
- $end(G)$: the synchronization at the end of segment G;
- $reach(l_1, l_2)$: the set of instructions on all *simple paths* from l_1 to l_2 in the CFG. A simple path has no repeated instructions.

Each segment has some unique instructions cloned by Algorithm 1; $segment(l)$ returns the containing segment for these instructions, or $None$ otherwise. We define a *partial order* over these instructions as follows: l_1 *happens-before* l_2, or $l_1 \prec l_2$, if (1) both l_1 and l_2 are synchronizations, and l_1 comes earlier than l_2 in the schedule; (2) $segment(l_1) = segment(l_2)$, and l_1 comes earlier in the segment; or (3) $\exists l_3$, s.t. $l_1 \prec l_3$ and $l_3 \prec l_2$. We say $G_1 \prec G_2$ if $end(G_1) \prec begin(G_2)$. Two instructions (segments) are concurrent if there is no happens-before between them.

At the top level, Algorithm 2 is similar to classic def-use analysis. CaptureConstraints takes a use u, calls PotentialDefs to get a set of potential defs, and, for each *live* def not killed, adds an equality constraint between the value used and the live def.

PotentialDefs illustrates how we implement the refinements enabled by schedule specialization. It processes a use u only if $segment(u)$ is not $None$, *i.e.*, u is unique to a segment. It searches for defs only in the instructions unique to each segment. To account for shared data access, it searches each thread t_d for a def of u. If t_d is the thread containing u, we search backwards from u for the latest dominator in CFG that accesses the same location. Otherwise, we search backwards from the latest synchronization s in t_d that happens-before u, and locate the latest dominator of s in CFG that accesses the same location. This dominator of s is essentially an inter-thread dominator of u. The result of PotentialDefs contains at most one def from each thread. In addition, it contains at most one store and possibly many loads (because we treat loads as defs).

Algorithm 2: Capture Constraints on Memory Locations

```
CaptureConstraints(u)
    foreach d ∈ PotentialDefs(u) do
        if not MayBeKilled(d, u) then
            AddConstraint("value(u) = value(d)")
PotentialDefs(u)
    defs ← ∅
    G_u ← segment(u)
    p ← loc(u)
    if G_u ≠ None then
        foreach thread t_d do
            if thread(G_u) = t_d then
                defs ← defs ∪ LatestDef(u, p)
            else
                G_d ← latest segment of t_d s.t. G_d ≺ G_u
                defs ← defs ∪ LatestDef(end(G_d), p)
    return defs
LatestDef(l, p)      // returns the latest intra-thread definition
    d ← l
    repeat
        d ← ImmediateDominator(d);
    until d = None or MustAlias(p, loc(d))
    if d ≠ None then return {d}
    else return ∅
MayBeKilled(d, u)
    G_d ← segment(d)
    G_u ← segment(u)
    if G_d = G_u then
        return MayStore(reach(d, u), u)
    foreach segment G s.t. begin(G) ≺ end(G_u) and
    begin(G_d) ≺ end(G) and G ≠ G_d and G ≠ G_u do
        if MayStore(G, u) then return true
    return MayStore(reach(d, end(G_d)), u) or
    MayStore(reach(begin(G_u), u), u)
MayStore(L, u)
    foreach store l ∈ L do
        if MayAlias(loc(l), loc(u)) then return true
    return false
```

Function MayBeKilled checks whether def d is killed by any instruction that may store to $loc(u)$ and execute after d and before u. It considers all instructions, not just the unique ones, for soundness. If d and u are in the same segment, *MayBeKilled* simply checks the instructions between d and u in the CFG. Otherwise, it checks instructions in any segment that may (1) begin no later than the end of u's segment G_u and (2) end no earlier than the begin of d's segment G_d. Note that these segments include not only the ones concurrent to G_u or G_d, but also any segment G s.t. $G_d \prec G \prec G_u$.

The above algorithm to collect constraints on memory locations needs alias analysis to determine whether two pointers must or may alias. We answer these queries again using STP. The question "whether v_1 and v_2 may alias" is rephrased as "whether $v_1 = v_2$ is satisfiable," and "whether v_1 and v_2 must alias" is rephrased as "whether $v_1 = v_2$ is provable." Since constraint solving may take time, we also ported `bddbddb` [36], a faster but less precise alias analysis to our framework, by writing an LLVM front end to collect program facts into the format `bddbddb` expects. We always query `bddbddb` first before STP.

Once constraints are captured, we perform a series of analyses based on these constraints. One analysis infers which LLVM virtual register has constant values (recall that these registers are in SSA form). It first queries STP for an assignment of value v_j to each variable a_j. For each a_j, it then adds $a_j = v_j$ to the current constraints, and checks whether STP can prove the new constraints. If so, it replaces a_j with v_j, such as replacing variable `p` in Figure 1 with constant 2. Once variables are replaced with constants, we perform stock LLVM analyses, such as constant folding, dead code elimination, and loop unrolling, which automatically gain precision due to schedule specialization. These analyses may further simplify a program, so we iteratively run data-flow specialization and these stock analyses until the specialized program converges.

6. Implementation

We implement our framework within LLVM [1]. The specialization algorithms described in the previous section are implemented as an LLVM pass that specializes the LLVM bitcode representation of an original program into a new bitcode program. These passes are of 11,754 lines of C/C++ code, with 2,586 lines for control-flow specialization and 9,168 lines for data-flow specialization. The remaining of this section discusses several implementation issues.

6.1 Constructing Call Graph

For clarity, Algorithm 1 assumes a simple call graph where each function call has a unique callee and unique return address. However, the programs we analyze do use function pointers and exceptions, invalidating this assumption. To resolve a call to a function pointer, we query our alias analysis and call *TryDFS* on each possible callee. To handle exceptions, we pair up the LLVM instruction that unwinds the stack (`UnwindInst`) with the call instruction that sets an exception handler (`InvokeInst`).

6.2 Optimizing Constraint Solving

Data-flow specialization (§5.2) frequently queries STP, and these queries can be quite expensive. We thus create four optimizations to speed up constraint solving. First, when identifying which variables are constants, we avoid querying STP for each variable. Specifically, when we query STP to prove that a variable is constant, we remember the value assignment STP returns even if STP cannot prove the query. If a variable has two different values in these assignments, then this variable cannot be constant, so we do not need to query STP whether this variable is constant. This optimization alone speeds up our framework by over 10 times.

Second, we cache extensively. Recall that data-flow specialization runs Algorithm 2 and other analyses iteratively. Within each iteration, we cache all alias results to avoid redundantly querying STP. Across iterations, we cache "must" and "must-not" alias results and the def-use results because each iteration only adds more constraints and does not invalidate these precise results.

Third, we use a union-find algorithm to speed up equality constraints solving. The def-use constraints we collect are all equality constraints. When the number of STP variables involved in these constraints increases, the STP queries tend to run slower. To reduce the number of STP variables, we put program variables that must be equal in one equivalent class represented by a union-find data structure, and assign only one STP variable to each class.

Lastly, we parallelize constraint solving. The STP queries issued by our framework are mostly independent. Specifically, each run of Algorithm 2 on a use is completely independent of another. In addition, checking whether one variable is constant is often independent of checking another. We have built a parallel version of STP that speeds up our framework by roughly 3x when running on a quad-core machine.

6.3 Manual Annotations

Our framework supports three types of annotations to increase precision and reduce analysis time. First, users can provide a blacklist of synchronization statements to customize which synchronizations go into the schedules (§3). Second, users can write regular assertions to enable our framework to collect more precise constraints. For instance, they can write "`assert(lower_bound <= index && index <= upper_bound)`" to inform our framework the range constraints on `index`. Third, users can provide function summaries to reduce analysis time. Some functions in the evaluated programs are quite complex and contain many load and store instructions, causing our framework to collect a large set of constraints. However, these functions often have simple shared memory access patterns which users can easily summarize to speed up constraint solving. These summaries can be quite coarse-grained, such as "function `foo` writes an unconstrained value to variable `x`." Users write summaries by writing fake functions, which our framework analyzes instead of the original functions. By supporting annotations in forms of assertions and fake functions, we avoid the need to support an annotation language.

Of the 17 programs evaluated, we annotated only 4 programs. We blacklisted the lock operations in the custom memory allocator in `cholesky`. We annotated two loops in `raytrace` with assertions because they are not in LLVM-canonical form. (Currently our framework handles only canonical loops.) We summarized five functions: `image_segment` and `image_extract_helper` in `ferret`, `TraverseBVH_with_StandardMesh` and `TraverseBVH_with_DirectMesh` in `raytrace`, and `LogLikelihood` in `bodytrack`. These summaries range from 8 to 18 lines.

7. Applications

To demonstrate the precision of our framework, we build three powerful analyses. The first is a precise schedule-aware alias analyzer, built on top of our def-use analysis (§5.2). We chose to build an alias analyzer because alias analysis is crucial to many program analyses and optimizations. Our analyzer provides a standard MayAlias(p, q) query interface, making it effortless to switch existing analyses to our our alias analyzer. Under the hood, MayAlias first queries a coarse-grained, existing alias analysis; if this coarse-grained analysis returns true, MayAlias then queries our def-use analysis for precise answers. The existing alias analysis we used is the C version[3] of `bddbddb` ported to LLVM.

The second tool is a precise static race detector that detects races that may occur only when a schedule is enforced. The detection logic appears identical to classic race detectors: two memory accesses are a race if (1) they may alias, (2) at least one of the accesses is a store, and (3) they are concurrent. There are two key differences. First, ours uses schedule-aware alias analysis. Second, ours detects concurrent accesses w.r.t. a total synchronization order (§5.2), more stringent than the execution order constraints dictated by the synchronizations. These two differences enable our detector to emit no or extremely few false positives (§8.1). Multiple races

[3] The alias analysis we use makes several assumptions about the C programs it analyzes for soundness; these assumptions are described in [9].

flagged on a specialized program may map to the same race in the original program because control-flow specialization clones statements. Our race detector emits only one report in this case.

The third tool is a thread-sensitive path slicer. Intuitively, given a program path, *path slicing* removes from the path the statements irrelevant to reach a given target statement [21]. The algorithm for doing so tracks control- and data-dependencies between statements, and removes statements that the target does not control- or data-depend upon. As previous work describes, path slicing can be applied to many problems, such as post-mortem analysis [21], counter-example generation [21], and malicious-input filtering [15]. Our path slicer improves upon existing path slicing work [21] by tracking dependencies across threads. Dependencies may arise across threads due to data or control flow. For example, if thread t_0 stores to pointer p, and t_1 then loads from q which may alias p, then the load in t_1 data-depends on the store in t_0. Similarly, if different branches of a branch statement in t_0 may cause t_1 to load different values from a shared variable, then the load in t_1 control-depends on the branch statement in t_0. Our path slicer correctly tracks these dependencies by leveraging our precise alias analyzer. Another feature of our path slicer is that it can slice toward multiple targets.

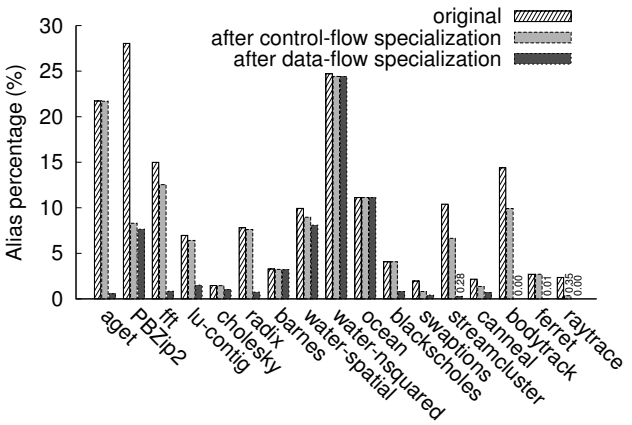


Figure 8: *Precision of the schedule-aware alias analysis.* Y axis represents the percentage of alias queries that return "may" responses. Lower bars mean more precise results. Each cluster in the figure corresponds to the results from one program. The three columns in a cluster represent the alias percentage when applying our analysis on the original program, the control-flow specialized program, and the data-flow specialized program.

8. Evaluation

We evaluated our schedule specialization framework on a diverse set of 17 programs, including `PBZip2` 1.1.5, a parallel compression utility; `aget` 0.4.1, a parallel `wget`-like utility; parallel implementations of 15 computation-intensive algorithms, 8 in SPLASH2 and 7 in PARSEC. We excluded 4 programs from SPLASH2 and 6 from PARSEC because we cannot compile them down to LLVM bitcode code, they use OpenMP [13] instead of Pthreads [34], data-flow specialization runs out of time on them, the compiled code does not run correctly in 64-bit environment, or our current prototype cannot handle them due to an implementation bug. All of the programs were widely used in previous studies [19, 25, 29, 38, 39].

We generated schedules for the programs evaluated using the following workloads: for SPLASH2 programs, we used the default arguments except that we ran them with four threads. These programs finish within 100 ms. For `PBZip2`, we compressed a 10 MB randomly generated text file. For `aget`, we downloaded a file of 1 MB from the internet. Unless otherwise specified, we ran all programs using four threads. This machine is a 2.8GHz Intel 12-core machine with 64 GB memory running 64-bit Linux 2.6.38. We compiled all programs using Clang and LLVM 2.9 with the default optimization level (often `-O2` or `-O3`) of each program.

In the remainder of this section, we first focus on two evaluation questions: (1) how much more precise the analyses become with schedule specialization (§8.1); and (2) what the overhead of our framework is (§8.2). We then present the bugs we detected with the precision provided by schedule specialization (§8.3).

8.1 Precision

We measured how our framework can improve the precision on the three analyses we built, the alias analyzer, the race detector, and the path slicer (§7). Our methodology is to apply these analyses on an original program, the control-flow specialized program, and the data-flow specialized program, and then compare the results.

Alias analysis precision. We quantified alias analysis precision by measuring *alias percentage*, the percentage of alias queries that return "may", instead of "must" and "must-not", responses because the latter two responses are precise. We selected the two pointers in each query from different threads to stress-test our alias analyzer because these pointers may appear to point to the same global array or the "same" thread-private heap or stack location, but are actually not aliases because most programs we evaluated access distinct memory locations in different threads. The total number of such queries can be large for programs that do many memory accesses, so we sampled some percentage of the queries to bound the total query time. The sampling ratio for `aget`, `PBZip2` and `fft` is $\frac{1}{50}$; for `water-spatial`, $\frac{1}{5000}$; and for `barnes`, `water-nsquared`, and `ocean`, $\frac{1}{50000}$. For all other programs, we processed all queries.

Figure 8 shows the alias precision results. For 12 programs (`aget`, `PBZip2`, `fft`, `lu-contig`, `radix`, `blackscholes`, `swaptions`, `streamcluster`, `canneal`, `bodytrack`, `ferret`, and `raytrace`), control-flow and data-flow specialization combined greatly improved precision. For instance, they reduced the alias percentage down to below 0.1% for `aget`. For 7 (`PBZip2`, `fft`, `swaptions`, `streamcluster`, `canneal`, `bodytrack`, and `raytrace`) of these 12 programs, control-flow specialization improved the alias analysis precision significantly. For instance, it reduced the alias percentage of `PBZip2` from 28.04% to 8.30%, a 70.4% reduction. The reason is that these programs allocate a fair amount of thread-private heap or stack data. Since the control-flow specialization clones thread functions and thus automatically provides thread sensitivity, our alias analyzer distinguishes accesses from different threads, achieving high precision. Although control-flow specialization alone did not significantly improve precision for the other 6 programs (`aget`, `PBZip2`, `lu-contig`, `radix`, `blackscholes`, and `ferret`) of the 12 programs, it created opportunities for data-flow specialization to improve precision. For the remaining 5 programs (`cholesky`, `barnes`, `water-spatial`, `water-nsquared`, and `ocean`), the reduction was not significant. This small reduction could be caused by the imprecision in our analyzer (*e.g.*, it is not path-sensitive), or real bugs in the original programs (§8.3). The mean reduction over all programs is 61.9%.

Race detection precision. We report the precision of our race detector here, and describe the detected bugs in §8.3. To evaluate its precision, we compared the number of false positives it emitted to a baseline race detector we built. This baseline detector uses the same definition of concurrent accesses (§7), except that it queries our `bddbddb` port instead of our alias analysis because our precise alias results can only be computed assuming a given schedule will be enforced. Since the total number of concurrent memory access pairs may be large, we used sampling for some benchmarks. The sampling ratio was $\frac{1}{50}$ for `PBZip2`, `fft`, `barnes`, `water-spatial`, and `ocean`; and $\frac{1}{5000}$ for `water-nsquared`. For all other programs, we checked all concurrent access pairs.

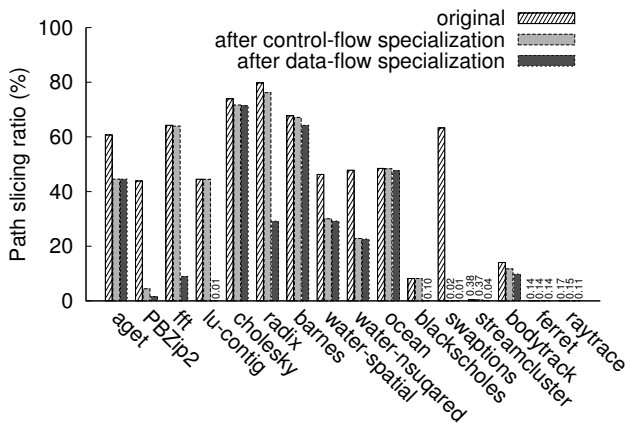


Figure 9: *Path slicing ratio on the original program, after control-flow specialization, and after data-flow specialization.*

Program	FP_{ours}	$FP_{baseline}$	*Races*
aget	0	72	1
PBZip2	0	125	32
fft	0	96	0
lu-contig	18	18	0
cholesky	7	31	0
radix	14	53	0
barnes *	369	370	n/a
water-spatial *	1799	2447	n/a
water-nsquared *	333	354	n/a
ocean *	292	331	n/a
blackscholes	0	3	0
swaptions	0	165	0
streamcluster	0	4	0
canneal	0	21	0
bodytrack	0	4	0
ferret	0	6	0
raytrace	0	215	0

Table 2: *Precision of the race detector.* FP_{ours} and $FP_{baseline}$ show the number of false positives for our detector and the baseline, respectively. *Races* show the true races detected. The four starred programs have a larger number of reports, so we conservatively treated all reports as false positives.

We counted the number of false positives for our detector, denoted by FP_{ours}, by inspecting its reports. Since the number of reports for `barnes`, `water-spatial`, `water-nsquared`, and `ocean` is large, we inspected a random selection of 20 reports, and found that they were all false positives. We thus conservatively treated all reports from these programs as false positives. We counted the number of false positives for the baseline detector, denoted by $FP_{baseline}$, by computing $R_{baseline} - R_{ours} + FP_{ours}$ where R is the number of reports. This formula works because our detector is more precise than the baseline detector, and a report flagged only by the baseline detector must be a false positive.

Table 2 shows the false positive comparison results for all 17 programs. For 10 of them, the precision of framework enabled our detector to reduce the number of false positives to 0. The reduction for `cholesky` and `radix` is also large. For `lu-contig`, `barnes`, `water-nsquared`, `water-spatial`, and `ocean`, our race detector reported slightly fewer races than the baseline; some of these results are expected based on our alias precision results. The mean reduction over all programs is 69.0%.

Table 2 also shows the number of true but benign races detected. (We present the harmful races in §8.3.) Our detector found 1 race on variable `bwritten` in `aget` and 32 races on variable `AllDone`, `NumBlocks`, and `OutputBuffer` in `PBZip2`. Without the precision of our framework, users may have to inspect hundreds of reports before finding the true races.

Program	LOC	Sched	Cons	Use	CF (s)	DF (s)
aget	866	219	2667	720	1.4	1551.7
PBZip2	9869	158	1382	480	3.1	973.1
fft	877	98	1122	277	1.5	113.8
lu-contig	904	48	824	229	1.4	65.0
cholesky	3962	42	1550	1302	2.3	1967.9
radix	919	112	1016	330	1.5	42.5
barnes	2234	5555	1605	19280	5.4	1968.1
water-spatial	1958	1037	14572	5008	2.8	313.4
water-nsquared	1620	4768	48023	14530	4.1	468.8
ocean	2958	5709	66253	18580	6.8	1795.0
blackscholes	1264	51	482	130	1.2	38.3
swaptions	1094	15	215	105	1.3	9.7
streamcluster	1765	90	840	216	1.5	834.6
canneal	2794	31	535	311	5.4	96.9
bodytrack	7696	381	998	240	1.7	20.9
ferret	10765	153	439	118	1.0	35.3
raytrace	13226	87	13468	417	1.4	5397.8

Table 3: *Specialization time.* **LOC** shows the lines of code in each program. The LOC of `PBZip2` includes the bzip2 compression library. We show the time spent in specializing control flow (**CF**) and data flow (**DF**). Since specialization time are affected by the schedule, constraints, and queries, we also show the schedule length (**Sched**), the number of constraints (**Cons**), and the number of uses in the def-use analysis (**Use**).

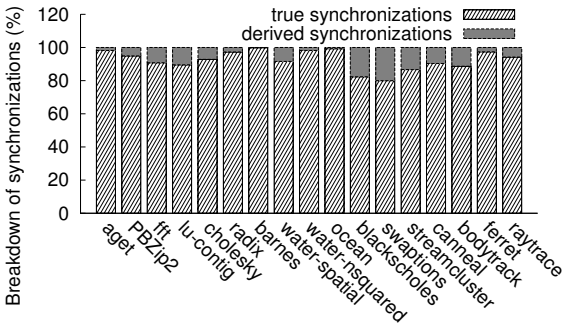


Figure 10: *True vs. derived synchronizations in schedules.*

Path slicer precision. To quantify the precision of our path slicer, we measured *slicing ratio*, the percentage of statements that remain in an execution trace. Figure 9 compares the precision of path slicing on the original programs and the specialized programs.[4] Control-flow specialization largely reduced the slicing ratio for `PBZip2`, `aget`, `swaptions`, `water-spatial`, and `water-nsquared`. For instance, it reduced the slicing ratio for `PBZip2` from 43.84% to 4.40%, a 89.97% reduction. Data-flow specialization further reduced the ratio for `PBZip2`, `swaptions`, `blackscholes`, `streamcluster`, `fft`, `lu-contig` and `radix`. For instance, it reduced the slicing ratio for `fft` from 64.25% to 8.95%, a 86.08% reduction. The ratio reduction for `cholesky`, `barnes`, and `ocean` was relatively small because our alias analysis sometimes reports may-alias for pointers accessed in different threads, causing more instructions than necessary to be included in the slices. The mean reduction over all programs is 48.7%.

8.2 Overhead

Table 3 shows the time specializing the control and data flow for each program. Control-flow specialization is much faster than data-flow specialization because it does not require expensive constraint-based analysis. Data-flow specialization typically finishes within minutes or hours. This time is correlated with the size of the schedule, the number of constraints collected, and the number of constraint-solving queries.

We also measured the number of derived synchronizations, *i.e.*, the calls to functions may transitively do synchronizations to re-

[4] We obtained the slicing results from an earlier prototype of our framework.

Program	Position	Bug	Effect
`aget`	Download.c: 87-95	`rbuf` may not have the byte sequence `\r\n\r\n`. `aget` needs an extra bound check.	race, program crash, or large data corruption
`aget`	Download.c: 98-99	The bound check should be `td->offset + dr - i > foffset`, not `dr - i > foffset`. The size passed to `pwrite` should be `foffset - td->offset`, not `foffset - i`.	race, or program crash
`aget`	Download.c: 99,101,113,115	`aget` should check the return value of `pwrite`.	race, program crash, or small data corruption
`aget`	Download.c:111	`aget` should check the return value of `recv`.	race, program crash, or small data corruption
`radix`	radix.C:148	`radix` should check variable `radix` against its upper bound 4096.	race, program crash.
`radix`	radix.C:159	`radix` should check `num_keys` against its upper bound 262144.	race, or program crash
`fft`	fft.C:162	`fft` should check `log2_line_size` against its upper bound 4.	program crash (before a race may occur)

Table 4: *Bugs found.*

solve call-stack ambiguity (§5.1). If this number is large, the overhead to record schedules and specialize programs may increase. Figure 10 shows that, for every program evaluated, the majority of the synchronizations in the schedule are still true synchronizations. Therefore, including derived synchronizations in the schedules does not incur significant overhead.

8.3 Bugs Found

The precision of our framework helped detect 7 previously unknown bugs in the evaluated programs. This result is particularly interesting considering that the evaluated programs have been well checked in previous work [19, 25, 29, 38, 39].

These bugs were typically detected as follows. Our analyses flagged two memory accesses from different threads as potential aliases or races, even though they should not be. We initially thought these "false positives" were due to the imprecision of our analyses and inspected them. Specifically, we queried STP for a solution to make the two pointers accessed identical.

Surprisingly, many of these "false positives" turned to be real bugs. A common cause is that an input variable is used as an array index without being checked against the upper or lower bounds of the array or the partition of the array assigned to a thread. Such off-bound accesses may indeed cause different threads to race, and our analyses thus flagged them. We detected 7 such bugs in `aget`, `radix`, and `fft`, which are shown in Table 4. These bugs may cause races, program crashes, or, worse, file data corruption. We manually verified these effects by running the buggy benchmarks on the bug-inducing inputs generated by STP. (The results presented in the previous two subsections are from the patched programs because we do not want these bugs to pollute our evaluation.)

9. Related Work

Slicing. Slicing techniques can remove irrelevant statements or instructions. Program slicing [35] does so on programs, dynamic slicing [6, 40] on dynamic execution traces, and path slicing [21] on (potentially infeasible) paths. Precondition slicing [15] is similar to path slicing with improved precision by incorporating dynamic information into the analysis.

Our technique to specialize a program toward a schedule differs from these slicing techniques because it takes as input both a program and a schedule, and outputs a specialized program that can actually run. Moreover, our technique does not merely remove statements; instead, it may transform a program by cloning statements when specializing the control flow of the program, replacing variables with constants when specializing the data flow, *etc*.

Deterministic multithreading. Several recent systems [8, 10–12, 16–18, 24, 28] eliminate nondeterminism due to thread interleaving; our TERN [16] and PEREGRINE [17] further reduce input nondeterminism. These DMT systems are not designed to facilitate static analysis. For instance, all existing DMT systems except TERN and PEREGRINE compute schedules online without storing them explicitly, making it difficult to analyze a program w.r.t. these implicit schedules. Moreover, although several DMT systems constrain a program to always use the same schedule for the same input, they may force the program to use a different schedule when the input changes slightly. This *instability* [16] not only aggravates input nondeterminism, but also largely prevents amortizing the static analysis cost in schedule specialization.

Nonetheless, our framework may use one of these systems to enforce schedules. Indeed, our framework leverages our PEREGRINE system, which explicitly stores and reuses schedules. Although PEREGRINE used a technique similar to the specialization algorithms described in §5, our PEREGRINE paper [17] described the technique mainly from a user's perspective, and presented no schedule specialization framework nor detailed algorithms.

Program specialization. Program specialization can specialize a program according to various goals [14, 20, 22, 27, 31]. For instance, it can specialize according to common inputs [14, 27]. The specialized programs can then be better optimized. Unlike previous work, our framework specializes a program toward a set of schedules, thus allowing stock analyses and optimizations to run on the specialized programs. To the best of our knowledge, we are the first to specialize a program toward schedules.

10. Conclusion and Future Work

We have presented *schedule specialization*, an approach to analyze a multithreaded program over a small set of schedules for precision, and then enforce these schedules at runtime for soundness. We have built a framework that specializes a program into a simpler program based on a schedule, so that the resultant program can be analyzed with stock analyses. Our framework provides a precise schedule-aware def-use analysis, enabling many powerful applications. Our results show that our framework can drastically improve the precision of alias analysis, path slicing, and race detection.

In our future work, we plan to leverage the precision provided by our framework to build precise error detectors, post-mortem analyzers, verifiers, and optimizers for multithreaded programs. In addition, we believe a similar specialization approach can improve analysis precision for sequential programs, too. In general, static analysis over all possible executions may be imprecise. To improve precision, we may perform static analysis over only a small set of executions (*e.g.*, the most common executions) and, if necessary, resort to dynamic analyses for the other executions. We believe this direction will face many interesting precision, soundness, and overhead tradeoffs, which we will investigate.

Acknowledgement

Alex Aiken, Stephen Edwards, Roxana Geambasu, Martha Kim, Eric Powders, and the anonymous reviewers provided many helpful comments, which have substantially improved the content and presentation of this paper. We thank Huayang Guo for LaTeX help. This work was supported in part by AFRL FA8650-11-C-7190, FA8650-10-C-7024 and FA8750-10-2-0253; and NSF CNS-1117805, CNS-1054906 (CAREER), CNS-1012633, and CNS-0905246.

References

[1] The LLVM compiler framework. http://llvm.org.

[2] The Princeton application repository for shared-memory computers (PARSEC). http://parsec.cs.princeton.edu/.

[3] Parallel BZIP2 (PBZIP2). http://compression.ca/pbzip2/.

[4] Stanford parallel applications for shared memory (SPLASH). http://www-flash.stanford.edu/apps/SPLASH/.

[5] STP Constraint Solver. https://sites.google.com/site/stpfastprover/.

[6] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation (PLDI '90)*, pages 246–256, 1990.

[7] Apache Web Server. http://www.apache.org.

[8] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[9] D. Avots, M. Dalton, V. B. Livshits, and M. S. Lam. Improving software security with a C pointer analysis. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*, pages 332–341, May 2005.

[10] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: a compiler and runtime system for deterministic multithreaded execution. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 53–64, Mar. 2010.

[11] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, pages 1–16, Oct. 2010.

[12] E. Berger, T. Yang, T. Liu, D. Krishnan, and A. Novark. Grace: safe and efficient concurrent programming. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '09)*, pages 81–96, Oct. 2009.

[13] O. A. R. Board. OpenMP application program interface version 3.0, May 2008.

[14] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Proceedings of the 20th Annual Symposium on Principles of Programming Languages (POPL '93)*, pages 493–501, 1993.

[15] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: securing software by blocking bad input. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 117–130, Oct. 2007.

[16] H. Cui, J. Wu, C.-C. Tsai, and J. Yang. Stable deterministic multithreading through schedule memoization. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[17] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Oct. 2011.

[18] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: deterministic shared memory multiprocessing. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 85–96, Mar. 2009.

[19] Q. Gao, W. Zhang, Z. Chen, M. Zheng, and F. Qin. 2ndStrike: towards manifesting hidden concurrency typestate bugs. In *Sixteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '11)*, pages 239–250, Mar. 2011.

[20] R. Glück and J. Jørgensen. Efficient multi-level generating extensions for program specialization. In *Proceedings of the 7th International Symposium on Programming Languages: Implementations, Logics and Programs*, pages 259–278, 1995.

[21] R. Jhala and R. Majumdar. Path slicing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI '05)*, pages 38–47, 2005.

[22] J. Jørgensen. Generating a compiler for a lazy language by partial evaluation. In *Proceedings of the 19th Annual Symposium on Principles of Programming Languages (POPL '92)*, pages 258–268, 1992.

[23] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[24] T. Liu, C. Curtsinger, and E. D. Berger. DTHREADS: efficient deterministic multithreading. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Oct. 2011.

[25] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Thirteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '08)*, pages 329–339, Mar. 2008.

[26] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *Proceedings of the 29th Annual Symposium on Principles of Programming Languages (POPL '02)*, pages 128–139, 2002.

[27] V. Nirkhe and W. Pugh. Partial evaluation of high-level imperative programming languages with applications in hard real-time systems. In *Proceedings of the 19th Annual Symposium on Principles of Programming Languages (POPL '92)*, pages 269–280, 1992.

[28] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 97–108, Mar. 2009.

[29] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 25–36, Mar. 2009.

[30] K. Poulsen. Software bug contributed to blackout. http://www.securityfocus.com/news/8016, Feb. 2004.

[31] T. Reps and T. Turnidge. Program specialization via program slicing. In *Proceedings of the Dagstuhl Seminar on Partial Evaluation*, volume 1101, pages 409–429. Springer-Verlag, 1996.

[32] M. Ronsse and K. De Bosschere. Recplay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, 1999.

[33] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*, pages 182–195, June 2000.

[34] The Open Group and the IEEE. POSIX.1-2008. http://pubs.opengroup.org/onlinepubs/9699919799/, 2008.

[35] M. D. Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, 1979.

[36] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)*, pages 131–144, June 2004.

[37] J. Yang, A. Cui, J. Gallagher, S. Stolfo, and S. Sethumadhavan. Concurrency attacks. Technical Report CUCS-028-11, Columbia University.

[38] W. Zhang, C. Sun, and S. Lu. ConMem: detecting severe concurrency bugs through an effect-oriented approach. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 179–192, Mar. 2010.

[39] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps. ConSeq: detecting concurrency bugs through sequential errors. In *Sixteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '11)*, pages 251–264, Mar. 2011.

[40] X. Zhang and R. Gupta. Cost effective dynamic program slicing. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)*, pages 94–106, 2004.