ShortCut: Accelerating Mostly-Deterministic Code Regions

Xianzheng Dou, Peter M. Chen, and Jason Flinn
University of Michigan

Abstract

Applications commonly perform repeated computations that are mostly, but not exactly, similar. If a subsequent computation were identical to the original, the operating system could improve performance via memoization, i.e., capturing the differences in program state caused by the computation and applying the differences in lieu of re-executing the computation. However, opportunities for generic memoization are limited by a myriad of differences that arise during execution, e.g., timestamps differ and communication yields non-deterministic responses. Such difference cause memoization to produce incorrect state.

ShortCut generically accelerates *mostly-deterministic* computation by partial memoization. It creates a program, called a slice, that modifies the state diff to account for variation in a subsequent computation. ShortCut learns which inputs, data flows and control flows are likely, and makes assumptions about possible values for each during slice generation. Assuming only likely values rather than allowing all possible values makes complex slice generation feasible and slice execution much faster. Slices are *self-verifying*; they include predicates that verify all assumptions made during a subsequent execution. When these verifications succeed, the slice is guaranteed to produce a correct modification. If a verification fails, ShortCut transparently rolls back the slice execution and runs the non-memoized computation. Users see no difference between normal, memoized, and rolled-back execution.

ACM Reference Format:
Xianzheng Dou, Peter M. Chen, and Jason Flinn. 2019. ShortCut: Accelerating Mostly-Deterministic Code Regions. In *ACM SIGOPS 27th Symposium on Operating Systems Principles (SOSP '19), October 27–30, 2019, Huntsville, ON, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3341301.3359659

1 Introduction

Applications often perform repeated computations that are mostly, but not exactly, similar. For example, Web servers perform many of the same actions when serving similar dynamic content, parallel stages in data pipelines have similar behavior, and application startup code does similar initialization. These mostly-similar computations consume resources, increase latency, and decrease throughput.

Systems currently use generic approaches such as memoization and starting from a checkpoint to accelerate *identical, deterministic* computation. Unfortunately, most code regions of reasonable size are neither identical nor deterministic, e.g., due to reading the time, external communication, differing thread schedules, etc.

Lacking generic options, application writers hand-craft custom optimizations such as pre-loading class files, precompiling headers, or running fast-paths for common executions. While such optimizations are often required for reasonable performance, application-specific solutions are time-consuming to develop, and the resulting code is often complex, buggy, and hard to maintain. Similar to how programmers prefer generic compiler optimizations to hand-crafted code optimizations, generic performance optimizations are preferable to hand-crafted solutions.

ShortCut is an OS-level solution that uses *partial memoization* to generically accelerate mostly-deterministic code regions without requiring application modification or source code. A region may be any contiguous sequence of instructions executed by an application. For each accelerated region, ShortCut transparently captures a *diff* by observing a *reference execution* of the region. A diff is the changes made to memory and register values and the system calls executed. ShortCut generates a program that modifies the diff to account for non-determinism and variant inputs that arise in subsequent executions of the region. This program, termed an *executable slice* [53, 57], consists of the instructions in the region that depend on any input that may change; typically, the slice executes many fewer instructions than the original execution.

To accelerate a subsequent execution, ShortCut runs the slice with the correct inputs, modifies the diff, and applies the modified diff to the execution. The computation is accelerated, and the program and kernel state at the end of the region are identical to what would have been generated by running the original execution.

ShortCut introduces a novel combination of three programming language and operating system techniques to make partial memoization feasible for complex applications: dynamic slicing [3], predicated analysis [19], and transparent

SOSP '19, October 27–30, 2019, Huntsville, ON, Canada
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6873-5/19/10...$15.00
https://doi.org/10.1145/3341301.3359659

rollback-recovery [23]. In addition, ShortCut uses deterministic record-and-replay to move time-consuming program analysis off of the critical path [7].

Several insights underpin the design of ShortCut. First, current approaches to slicing (which are used for debugging, etc.) construct slices that produce correct results for *all* possible inputs and code paths. This leads to large slices that produce minimal benefits. Further, generating such slices requires complex static analysis that does not scale to the large code regions (i.e., billions of instructions) that we target. Instead, ShortCut introduces the idea of *predicated slices* that embed a set of assumptions about how inputs, paths, and memory accesses may differ. Predicated slices are *self-verifying*: they check predicates as they run to verify all assumptions. If all predicates succeed, the slice is guaranteed to produce a correct modification. If a predicate fails, memoization is not used. Predicates substantially reduce slice size and make slice execution faster; they also let slice generation use static, rather than dynamic, analysis, which is essential to scaling analysis to large, complex code regions.

Second, ShortCut uses rollback-recovery to allow predicates to be verified as the slice executes rather than before execution begins. If a slice execution externalizes state by, e.g, making a system call or communicating with another process, the application is in an undefined state when a predicate fails. ShortCut restores the program state to the values at the beginning of slice execution, then runs the original (non-memoized) execution. It guides the restarted execution to make the same external actions as the slice and suppresses duplicate external effects, e.g., system calls [22].

ShortCut must observe several executions to learn, e.g., which initial values are likely to vary and which control paths are likely to be taken. An obvious approach to generating these observations is profiling a large set of sample runs. However, our third insight is that *incremental refinement*, in which ShortCut learns from failed memoization attempts is a much better approach. In a profile set, different executions exhibit *internal non-determinism*; the different executions have different thread schedules, read data in different-sized chunks, etc. The internal non-determinism causes cascading divergences in data and control flow that makes comparing two executions exceedingly difficult.

Instead, ShortCut starts with a single *reference execution*. It accelerates subsequent executions by running a slice that mostly eliminates such internal non-determinism: e.g., it follows the same thread schedule and reads data in the same-sized chunks from the network when feasible. If a predicate fails, ShortCut learns from failure by *refining* the slice, e.g., by allowing more inputs to vary or more paths to be taken, so that subsequent executions are more likely to succeed.

The program regions that benefit from ShortCut share several characteristics. First, they have at least several milliseconds of user-level CPU usage, which allows the benefit realized from partial memoization to exceed the fixed costs of loading and running a slice. Second, they are mostly deterministic, which means that the region is often executed with similar inputs and the control flow within the region does not diverge substantially from execution to execution. Our evaluation shows that application startup and request processing loops are good candidates for ShortCut optimization.

This paper makes several contributions:

- It articulates the goal of generic, system-level partial memoization.
- It introduces predicated slices as a method for accelerating mostly-deterministic code regions.
- It shows how predication and dynamic analysis let slices scale to code regions of unprecedented size and complexity.
- It shows how transparent recovery allows predicate evaluation at any point in slice execution.
- It proposes incremental refinement as a more effective alternative to profiling for generating slices.

We have applied ShortCut to a wide variety of applications with significant regions of mostly-similar computation[1]. In our benchmarks, ShortCut speeds up Web/PHP throughput by 392–493%, package compilation by 43–54%, and startup of user-facing applications (GNU Emacs, GIMP, Java/HDFS and a custom constraint solver) by a factor of 6–76. Further, predicates need not always succeed. In fact, across all benchmarks, predicates must succeed only 3–28% of the time for ShortCut to improve average performance.

2 Overview

In this section, we summarize ShortCut's design and motivate our implementation choices.

As a starting point, consider how the OS could support generic memoization of completely deterministic computation. Given a computation to memoize (which we call the *reference execution*), the OS could observe program state (register and memory values) modified by the computation. When the same computation occurs later, instead of actually performing the computation, the system applies the state diff to the program and jumps to the end of the memoized region.

There are two correctness requirements. First, the inputs to the memoized computation must be the same in the reference and subsequent executions; these include the initial values of all memory and registers read by the computation. The system could verify that any input seen during the reference execution is the same during the subsequent execution. If any of these verifications (called *predicates*) fail, the system simply runs the normal execution. Second, the memoized region must be deterministic, e.g., the region could be single-threaded and make no system calls. In practice, these restrictions are onerous, and the applicability of generic memoization to general-purpose computation is limited.

[1] The source code is available at github.com/shortcut-sosp19/shortcut

The goal of ShortCut is to substantially broaden the applicability of memoization by applying it to not just *identical* regions of computation, but also to regions that are *mostly similar*. To support partial memoization, we wish to allow some of the inputs to the computation to differ between the reference and subsequent executions, and we wish to handle the non-determinism that arises.

Given a set of inputs that are expected to differ in subsequent executions, ShortCut produces a *forward executable slice* during memoization. This is a small program that includes only the computation in the memoized region that depends on those inputs. On a subsequent execution, after applying the diff to the program state, it runs the slice to modify the state with the values that would have been derived by running the computation with the new inputs.

As an example, consider a reference execution that, along with many other actions, reads the time of day, stores the value at memory location A, adds 100 ms to the value, and stores the sum in memory location B. Generic memoization would be incorrect for this execution since the values at A and B would reflect the time the reference execution ran, not the time the subsequent execution actually ran. ShortCut produces a slice that only reads the time and modifies the program state at A and B with the correct values when each subsequent execution runs.

Unfortunately, producing such a slice can be very complex for large, general-purpose code regions. It is typically infeasible to produce a slice that correctly handles *all* possible executions that depend on *any* divergent input. For example, the difference in input values may affect the control flow or the order of thread execution. Inputs may be used as indexes into arrays so that the memory locations accessed vary. Values derived from divergent inputs may be passed to subsequent system calls in the memoized regions, and those system calls may produce different outputs as a result. The common approach to generating such slices, static analysis, simply does not scale to the size of regions ShortCut targets (billions of instructions) or work effectively on complex binary executables (a requirement if memoization is to be done generically by the OS).

One insight guiding the ShortCut design is that the slice does not need to handle all possible subsequent executions if it can be *self-verifying*, i.e., if it can embed predicates that verify whether or not it will produce a correct modification for a subsequent execution. For example, rather than produce a slice that handles all possible control flows, ShortCut produces a slice that is correct for a few, common control flows, and it inserts predicates in the slice to verify that the slice follows one of those control flows.

At first glance, all such verifications must be performed before the slice executes because the failure of a verification while executing the slice would leave the program in an undefined state. This requirement would be too onerous. For example, consider a control flow predicate that tests whether register `eax` has a value less than 10. It is trivial to evaluate this predicate at the branch point during slice execution (i.e., add a jump to a predicate failure routine if `eax >= 10`). However, evaluating this predicate before the slice begins is potentially intractable. The branch may be millions of instructions in the future, `eax` could depend on thousands of distinct instructions, and the chain of dependencies could be complex, involving aliasing, etc. Essentially, evaluating the predicate could require executing many of the preceding instructions in the slice anyway.

The second key insight guiding the ShortCut design is that rollback-recovery allows verification during slice execution. Rollback-recovery [23] is generic system support for restoring the program to an earlier state: this includes restoring all internal modifications (e.g., memory and register values), and suppressing duplicate external actions (e.g., system calls) made by the recovered execution.

If a verification fails during slice execution, ShortCut rolls the execution back to the start of the memoized region, then re-executes the full (non-memoized) execution. External observers (e.g., users and other processes) do not see any difference other than performance between a non-memoized execution, one accelerated using ShortCut, and one in which a verification fails, the slice is rolled back, and a full execution is performed.

Importantly, self-verifying slices and rollback-recovery allow ShortCut to use dynamic analysis, rather than static analysis, to generate slices. Dynamic analysis can derive a slice by observing a small set of executions as they run. As our results show, it can easily scale to code regions of billions of instructions and does not require source-code availability.

Generating a slice requires ShortCut to determine a set of inputs to the computation that are likely to vary in subsequent executions. Our initial approach to determining this set was to profile several executions and observe how they differ. Unfortunately, this approach worked poorly because these executions often had considerable internal non-determinism, e.g., they followed different thread schedules or read data from the network in different-sized chunks. In other words, natural internal non-determinism caused both data flow and control flow to diverge too much to compare executions.

Instead, ShortCut uses *incremental refinement*, in which it learns from its mistakes, to generate slices. Initially, it generates a slice based on a single reference execution. Only inputs that will clearly differ in subsequent executions (e.g., the time of day or process identifier) are allowed to vary; all other inputs are verified to be the same as the reference execution. If a verification fails during a subsequent execution, ShortCut refines the slice, e.g., by allowing inputs that led to the failure to vary during slice generation. Thus, if all verifications pass, the computation is accelerated. If a verification fails, the failure result provides profile data that allows ShortCut to generate better slices in the future. Importantly, because the slice is created to be similar to the reference execution, many sources of internal non-determinism are eliminated: e.g., the slice and the reference execution follow the same thread schedule and read data from the network in the same-sized chunks. Because natural variation due to internal non-determinism is greatly reduced, ShortCut is able to learn much more from failed slices generated during incremental refinement than it would from naturally-occurring executions in a profile set.

The last problem we faced in the design of ShortCut was how to hide the overhead of slice generation from users of the system. ShortCut uses deterministic record-and-replay to move slice generation out of the critical path. It records a reference execution using standard record-and-replay techniques [21] and then produces the slice by dynamically instrumenting a replay of that execution. Importantly, ShortCut can substantially reduce the cost of recording during subsequent executions (i.e., when executing a slice) because it records only the small set of non-deterministic inputs that differ between the slice execution and the reference execution rather than all non-deterministic inputs. These differences, combined with a replay log of the reference execution, allow ShortCut to reproduce the failed execution for subsequent dynamic analysis.

3 Implementation

We next describe the four major components of ShortCut: slice generation, slice execution, transparent recovery, and incremental slice refinement.

Slice generation, described in Section 3.1 dynamically analyzes a reference execution to partially memoize a region of computation. The outputs of slice generation are a state *diff* that specifies the memory and register values to change, and a *slice* that modifies the diff and makes system calls to modify state external to the process.

ShortCut uses Pin [40] to execute programs called Pintools that observe and modify the state of an x86 application as it executes. The slice generation Pintool adds logic to each x86 instruction that observes its inputs and outputs, e.g., to determine which instructions depend on inputs that may change in subsequent executions so that these instructions can be included in the generated slice.

Dynamic binary instrumentation has high overhead, but it supports the complex operations we need for slice generation. To hide overhead, ShortCut uses deterministic replay to decouple analysis and the original application execution. When an application executes, it uses Arnold [20] to record the non-deterministic inputs to the application. Arnold can later replay the application on demand with Pin instrumentation; Arnold guarantees that the instrumented execution will execute the same instructions and have the same data values as the original execution for race-free programs. It also detects races that occur.

Slice execution is described in Section 3.2. When the application reaches the start of the memoized region, it traps to the OS, which applies the state diff and then loads and runs the slice. Predicate failures during slice execution makes a system call to initiate transparent recovery. The last instruction in the slice makes a system call that lets the OS unmap the slice, perform final cleanup, and set the instruction pointer to the end of the memoized region.

During transparent recovery, described in Section 3.3, ShortCut generates a *synthetic execution*, which is a replay log that produces the normal (non-memoized) execution of the region with the inputs seen by the rolled-back slice execution. It starts a new process replaying from the application state at the beginning of slice execution; the replay generates the correct memory and register state at the point the predicate failed. ShortCut swaps this memory state with that of the failed slice (which has the correct kernel state). ShortCut then resumes normal application execution with the original process and throws away the process used for recovery.

Incremental slice refinement, described in Section 3.4, allows ShortCut to learn from failure by broadening the slice to account for divergences that led a validation to fail when running the slice. Incremental slice refinement may specify which inputs are expected to diverge, which control flow paths are expected, and which memory addresses are expected to be accessed by relative loads and stores. These expected divergences are used by the slice generator to produce a new slice, e.g., one that would have not failed the observed predicate. Incremental refinement requires dynamic analysis, so it is implemented as a Pintool. ShortCut uses Arnold to execute the analysis offline.

3.1 Slice generation

Slice generation is an offline process that instruments a deterministic replay of a reference execution to produce one slice and one state diff associated with that slice. The start of the region to memoize is specified manually: this is either the first instruction after loading the executable or a specific system call made by the reference execution. The end of the region can also be specified manually (as a specific system call), or determined automatically by incremental refinement when a divergence occurs that is too large to handle.

Slice generation optionally takes as input a set of expected *divergences*, which are inputs, control flow paths, and memory accesses that are expected to differ in subsequent executions. The set of divergences is generated through incremental refinement, so it is initially empty when the reference execution is first analyzed.

ShortCut generates the diff by replaying the reference execution to the end of the region and recording the values that it writes to memory and registers. Note that simply comparing start and end memory and register values is insufficient, e.g., if the region writes a 1 to a location that happens to already contain a 1, that location would be omitted from the diff; which would lead to a missing modification if the location contains 2 during a subsequent execution. However,

```
1   // initially: x=1, y=2, z=3, A=[0,1,2,...]
2   v = x;
3   w = y + z;
4   rc = write(x);
5   t = time();
6   if (t > y)
7   write(checks out");
8   else
9   BigFunction();
10  b = A[y];
11  c = A[w];
```

(a) Reference execution

```
1   // only z might differ
2   check(x=1);
3   check(y=2); w = 2 + z;
4   rc = write(1); check(rc=4);
5   t = time();
6   check(t > 2);
7   write(checks out");
8
9
10
11  check(5<=w<=6); A[5]=5; A[6]=6; c=A[w];
```

(b) Slice execution

```
1   {z}
2   {z}
3   {w,z}
4   {w,z}
5   {t,w,z}
6   {t,w,z}
7   {t,w,z}
8   {t,w,z}
9   {t,w,z}
10  {t,w,z}
11  {c,t,w,z}
```

(c) Taint set

```
1   {}
2   {v=1}
3   {v=1}
4   {v=1,rc=4}
5   {v=1,rc=4}
6   {v=1,rc=4}
7   {v=1,rc=4}
8   {v=1,rc=4}
9   {v=1,rc=4}
10  {b=2,v=1,rc=4}
11  {b=2,v=1,rc=4}
```

(d) State diff

Figure 1. **Slice and diff generation example.** The first column shows the reference execution and the second the resulting slice. The third column shows how the taint set changes over time and the last shows how locations are added to the state diff. Assume that incremental refinement has revealed that w is likely to be 5 or 6.

as an optimization, if the region starts at the beginning of the program execution, ShortCut simply uses a checkpoint taken at the end of the region at the diff, which is correct because there are no pre-existing values.

ShortCut generates the slice by replaying the reference execution with a Pintool that implements a taint tracking [42] algorithm that identifies which *locations* (memory addresses, registers, and CPU flags) could potentially have different values in a subsequent execution, assuming all predicates up to that point have succeeded. ShortCut tracks taint at byte granularity by analyzing individual x86 instructions.

The generated slice is an x86 assembly program that is linked with libc and a support library to produce executable code. In general, the slice includes each system call made by the reference execution, so executing the slice will have the exact same effect on the kernel and external applications as executing the non-memoized version of the region.

The first two columns in Figure 1 shows a sample region to be memoized and the resulting slice (for clarity, we show pseudocode instead of x86 assembler code). The third column shows the taint set (i.e., the locations that depend on inputs that may vary). The final column shows the state diff (i.e., the locations changed by the reference execution); in the figure, we show only the locations that will not be modified by slice execution.

3.1.1 Slice inputs

Inputs to the memoized reference execution may come from: (1) initial memory and register values, (2) the registers and memory modified by system calls, (3) non-deterministic x86 instructions such as `rdtsc`, and (4) shared memory modified by other applications.

Each input is either *predicated* or *tainted*. A predicated input is always accompanied by a predicate in the slice that checks that the value of the input during subsequent executions is the same as that seen during the reference execution. Predication of initial memory and registers is done when the dynamic analysis first observes an instruction reading that location. In Figure 1, line 2 inserts a predicate that checks if x is equal to 1 (the value read by the reference execution). If a predicate check fails, the slice jumps to code that initiates transparent recovery (not shown).

Tainting an input means that the input may contain any value during subsequent executions. Initially, z is tainted because it is expected to differ based on previous incremental refinement results. Subsequently, the slice generator will include any instructions that depend on z in the slice, and it will taint any values that depend on a tainted value, such as w on line 3.

As an optimization, if the slice begins at application start, ShortCut only taints or predicates arguments and environment variables. It also predicates that the executable image has not changed to ascertain that other initial inputs are the same as those seen for the reference execution.

Registers and memory locations modified by system calls are also tainted or predicated. Some system call results are inherently non-deterministic: e.g., t returned by `time` in line 5 or file access times returned by `fstat`, etc. ShortCut always taints such results. Otherwise, it only taints the system call result if different values are expected based on incremental refinement. In line 4, the return code from write is not tainted (e.g., because it has not differed during previous executions). ShortCut taints memory buffers modified by system calls on a byte-by-byte basis, while all bytes in numeric values are tainted together. Values read from shared memory or modified by non-deterministic x86 instructions such as `rdtsc` are always tainted.

System calls such as `recv` non-deterministically return different numbers of bytes (e.g., depending on how much data has been received from the network). ShortCut handles this non-determinism in two ways. First, the ShortCut profiler may observe the system call returning different numbers of bytes and allow it to return any number of bytes up to a maximum of n bytes during slice execution. In this case, the first n bytes in the memory buffer populated by the system call are tainted, even if the system call returns less than n bytes. Second, ShortCut rewrites the inputs to the system call before passing them to the kernel so that no more than n bytes are returned. The kernel will buffer any excess bytes and, e.g., provide them on the next `recv` call. Our evaluation shows that this approach works will for TCP and similar stream protocols (e.g., for communication between GUI applications and the X server).

3.1.2 Taint tracking for x86 instructions

The slice generator inspects each x86 instruction during the reference execution, and performs three tasks. First, if the instruction has at least one tainted input, it is included in the slice; if no inputs are tainted, the instruction is omitted and the next two steps are skipped (e.g., line 2 is omitted from the slice because x is not tainted)

Second, the slice generator rewrites the instruction and replaces any non-tainted inputs with constants equal to the value of the input at the beginning of the instruction in the reference execution. For example, line 3 is included in the slice because z is tainted, and the untainted y is replaced with the constant 2 in the resulting instruction.

An untainted value will not differ during subsequent executions since it does not depend on any input that is allowed to vary (a prior predicate would have failed if an input it depends on had varied). Note that non-tainted locations are not guaranteed to have correct values during slice execution, so replacement with constant values is necessary.

Third, the slice generator taints all outputs of the instruction that depend on the tainted inputs. So, w is tainted after line 3.

ShortCut tracks taint for almost all x86 instructions, including floating point instructions, x86 SSE instructions, repeat string instructions, etc. The slice generator verifies completeness by ensuring that taint tracking is implemented for all instructions with a tainted input in the sliced region.

3.1.3 System calls

System calls are a special case: logically, any system call in the reference execution is added to the slice whether or not its inputs are tainted. This ensures that slice execution has the same external effects, including changes to kernel state, that non-memoized execution would have.

If a system call input is tainted, the slice uses the memory or register value from slice execution (this location holds the same value that would have been calculated by the non-memoized execution). Otherwise, the slice generator replaces the system call parameter with a constant (e.g., as in line 4). Since tainting is done at byte granularity, the decision about whether to replace with a constant is also done at byte granularity. This allows, for example, the output of a string that contains some unmodified values and some modified values such as the current time or process identifier.

3.1.4 Data flow divergences

Most x86 instructions that access memory can use base and index registers to specify a relative offset for the address accessed. For instance, consider the code in Figure 2. Line 1 sets register `eax` and line 2 uses it as the base register for a memory load at effective address `0x12345678`. Since the inputs to line 1 are not tainted, it will will not be included in the slice. Line 2 will be in the slice, but the contents of `eax` are undefined since line 1 was omitted. Therefore, the slice generator rewrites line 2 as shown in the second column, specifying the load address as a constant.

If a base or index register is tainted, the effective address calculated for subsequent executions may differ from that calculated by the reference execution. This is potentially incorrect since the slice may not have a correct value at that effective address. By default, the slice generator inserts a predicate prior to the load or store that checks that any tainted base or index register has the same value as during the reference execution. In line 3, the slice checks that the value loaded into `ebx` is `0x1111` (the value seen during the reference execution).

Since tainting means that a register's contents might vary during subsequent executions, and we have predicated that it should not vary from what have been observed, taints can be safely removed from the base and index registers afterwards; this important optimization reduces the number of subsequent predicates and simplifies the slice. For example, line 4 in Figure 2 does not need a predicate.

What if the memory address commonly varies from execution to execution? ShortCut detects variation during incremental refinement and allows a *data flow divergence* by specifying the instruction where divergence occurs, as well as a set or range of effective addresses that will be allowed.

When the slice generator encounters a specified instruction that reads memory, it first adds a predicate that checks the actual effective address is in the specified set or range. Next, it adds instructions that ensure each allowed address has a correct value. Tainted addresses already have correct values but untainted addresses must be set to the current value at that location in the reference execution. As an optimization, initialization is omitted in cases where the slice generator can guarantee that the untainted location already has the correct value. Finally, the slice generator adds the original instruction to the slice. It taints the outputs of this instruction since they may differ depending on which memory address is read.

For example, line 11 in Figure 1 has a tainted array index. Assume that incremental refinement has revealed that w is likely 5 or 6. The slice inserts a predicate for this assumption, initializes memory with values seen in the reference execution (the example assumes these locations are untainted), and performs the read. c is tainted because its value will depend on which location was read, even though neither location was tainted. In contrast, line 10 does not result in any slice code since the base, index, and memory location read are untainted.

Divergent writes are handled similarly. The slice generator adds a predicate to check that the store address falls within the specified set or range, and it ensures that all potential store addresses have correct values prior to executing the

```
mov eax, 0x12345678
mov ebx, DWORD PTR [eax]
mov ecx, DWORD PTR [ebx]
mov edx, DWORD PTR [ebx+0x4]
```

(a) Reference execution

```

mov ebx, DWORD PTR [0x12345678]
cmp ebx, 0x1111; jne fail

```

(b) Slice execution

Figure 2. **Base register example.** The first column shows the reference execution and the second the resulting slice.

instruction. After the instruction, all locations in the target range or set are tainted.

3.1.5 Control flow divergences

Initially, the slice generator inserts predicates to ensure that the slice follows the same control flow as the reference execution. At each branch or jump, it checks if the inputs to the instruction are tainted; e.g., it checks if any flag read by an x86 conditional branch is tainted. If all inputs are untainted, the control flow instruction is omitted from the slice as the same branch will be taken in all subsequent executions (because the prior predicates have succeeded). If an input is tainted, the slice generator inserts a predicate that ensures that the branch outcome is the same in the reference and subsequent executions. For example, line 6 in Figure 1 checks that the branch condition is true given one tainted input t and one untainted input y. The slice omits the branch as well as all instructions not on the path taken by the reference execution.

If incremental refinement observes that multiple paths are taken, it may specify a control flow divergence by giving one or more alternate paths of execution. For each such divergence, it specifies the instruction where divergence may occur, the instruction where the divergence ends (called the merge point), and allowed paths from the divergence point to the merge point.

Given a divergence, the slice generator adds alternate paths to the slice that differ from the one taken by the reference execution. It speculatively executes each alternate path from the conditional instruction to the merge point, adding code for each path to the slice. It then rolls back execution to the conditional instruction and executes the path taken by the reference execution. It inserts conditional instruction(s) in the slice to take the appropriate path based on the value of the tainted inputs during subsequent executions.

The slice generator tracks the set of locations modified along any path. At the merge point, it taints all such locations, and it inserts code at the end of each path to copy correct values into any previously untainted locations. Values in these locations can differ due to the implicit flow caused by executions taking different paths [59].

ShortCut limits control flow complexity by disallowing alternate paths that include a system call or synchronization instruction and allowing no more than 512 basic blocks in any alternate path. Such paths are not included in the slice, and ShortCut checks that they are not taken by subsequent executions.

3.1.6 Multithreading and signals

Arnold replay is equivalent to executing the replayed program on a single core with all context switches between threads occurring at system calls and synchronization operations [20]. There exists a total, deterministic order over all replayed instructions that is equivalent to the original multicore execution if the program is race-free [47]. Since alternate paths cannot span system calls or synchronization operations, the slice has the same sequential ordering over such operations as the replay of the reference execution.

Due to this property, slice generation safely replaces all synchronization such as pthread mutexes and low level locks with code that explicitly switches between threads when a slice instruction in one thread is followed by one in another thread. Peregrine [18] used a similar technique to support deterministic multithreading and noted substantial performance benefit.

After slice execution, mutexes and other synchronization objects may have incorrect values, so the slice generator appends code to the slice to perform compensating actions. For example, a thread that is holding a lock at the end of the region acquires the lock.

When feasible, slice execution delivers signals at the same point in the slice execution as the signal was originally delivered in the reference execution. If the signal arrives early, the kernel buffers the signal until the slice reaches the desired point in its execution. If the signal arrives late, the slice pauses for a few milliseconds to see if the signal will arrive. If ShortCut cannot deliver the signal at the desired point in the slice execution, it initiates transparent recovery.

3.1.7 Outputs

The slice produces correct outputs at the end of the region for all tainted memory addresses and registers. However, some locations may have been modified by slice execution and then became untainted. For example, a variable could first be assigned a tainted value and then subsequently set to zero. The first instruction would be in the slice, but the second instruction would be omitted. The slice writes the correct final value to each such location at the end of its execution.

If the reference execution writes to shared memory, the slice generator inserts an instruction that writes to that location even if the instruction inputs are untainted; in this case, the value written is a constant equal to the value written by the reference execution.

3.2 Running a slice

ShortCut runs a slice when the program reaches the first instruction in the memoized region. It evaluates any initial predicates. For example, if the slice starts at the beginning of the program, it validates that any untainted argument or environment variable are the same as in the reference execution. This is equivalent to running the slice only if a subset of the arguments and environment variables match. Similarly, if the slice starts at the beginning of a function, it validates all memory and register dependencies of the function, such as input arguments, global variables and static variables used by the function. If any initial validation fails, ShortCut lets the execution proceed normally. If all succeed, ShortCut first applies the diff by modifying memory and register values. Then, it executes a system call to have the kernel load the slice into a free location in the application address space (i.e., one that was never mapped during the reference execution). The kernel allocates a stack and sets the instruction pointer to the first instruction in the slice before returning.

Since the slice executes all system calls made by the reference execution, it makes the same modifications to kernel state and communicates the same way with external processes as would a normal execution of the memoized region.

On successful completion, the slice executes a system call to return control to the kernel. The kernel removes the slice and stack from the address space, and it restarts each application thread. The application runs normally from this point without further kernel support.

3.3 Transparent recovery

If a predicate verification fails during slice execution, the slice executes a system call to ask the kernel to initiate transparent recovery. Predicates may fail at any point during slice execution, but ShortCut cannot restart execution from any arbitrary point because the address space of the slice has only a small portion of the correct values that would exist during a normal execution.

A simple, but flawed, design would be for ShortCut to restart the application from the beginning of the region. Unfortunately, the slice will execute system calls that externalize output, e.g., write to files or communicate with other processes. A simple restart would create inconsistent external state, such as file modifications and ghost UI windows.

Rollback-recovery [23] using deterministic replay seems promising. With this technique, the kernel rolls back application state to a prior point, then replays the execution that fails, supplying the same non-deterministic inputs (called the *replay log*) seen by the failed execution and suppressing duplicate output produced by the replayed execution (in this case, the system calls made by the replayed process). Rollback-recovery can recreate the memory state at any point in a failed execution.

However, which execution do we replay? The reference execution has different inputs than those seen by the slice: replaying it will produce application memory inconsistent with state externalized by the slice. The slice execution is incomplete: replaying it will reproduce the same partial memory state we already have.

ShortCut's solution is to create a *synthetic execution* by constructing a new replay log that includes all non-deterministic inputs that would have been recorded by Arnold running the normal execution with the inputs seen by the slice. The simplest method to generate this log would be to have the slice record all of its non-deterministic inputs as it executes.

However, since the replay log of the slice is expected to be very similar to that of the reference execution, ShortCut optimizes recording by having the slice record only instances where the input it sees is different from that of the reference execution. If needed, ShortCut patches the reference execution's replay log with the recorded differences to produce the replay log that would have been recorded by the slice. Since recording overhead depends substantially on the amount of data recorded and because this optimization greatly reduces the size of data that needs to be recorded, this optimization significantly reduces recording overhead during slice execution in the common case where all predicates succeed.

As a further optimization, the slice is generated with code that writes any input that differs from that seen by the reference execution to a memory buffer. The buffer is only flushed to disk when full.

On predicate failure, the kernel invokes a user-level daemon to construct a synthetic replay log by patching the reference execution log with the buffered input values written by the failed slice. The kernel creates a new process and replays the synthetic log to the point where the predicate failed. Arnold replay suppresses all system calls made by the synthetic execution except those that affect process state, such as memory mapping and thread creation. This means that the replayed execution makes the same system calls with the same inputs provided by the original slice execution, but those system calls are not executed by the kernel. For example, if the slice sent a message to a remote server, then the replayed execution will try to send the same message, but the kernel will not actually send it. If the slice received the response, the replayed execution is supplied with the same response seen by slice execution in lieu of performing the receiving system call.

Note that the correct memory state for the application is now associated with the synthetic replay process, and the correct kernel state is associated with the slice process that failed. So, the kernel simply swaps the address space and registers of the two processes. The original slice process is allowed to execute from this point, and the application runs normally since it has correct state. The process used for the synthetic execution is destroyed.

From the perspective of the user or any external observer, there is no difference between normal execution, partially-memoized execution, and an execution that recovers from a failed predicate, except that partially-memoized execution is much faster and a recovered execution is slightly slower than normal execution.

3.4 Incremental slice refinement

ShortCut uses additional executions of the memoized region to refine the initial slice generated by the reference execution. For example, initially ShortCut may expect z to be 3 in 1a for all executions, so line 11 inserts a predicate that simply checks if w is 5 as w depends on z. However, if a subsequent execution reveals that z could also be 4, during the refinement process, the predicate in line 11 evolves to a range check for w. Refinement is based on dynamic analysis performed offline by running a Pintool to instrument previously recorded executions. Thus, users see no performance impact from refinement when running applications.

Recall that ShortCut allows three types of divergences when generating a slice: input, data flow, and control flow. Refinement is the process of deciding which divergences should be allowed for subsequent executions of the slice. Since refinement uses the past to predict the future, it will have both false positives and false negatives. ShortCut tolerates both types of errors because slices are self-verifying.

A false negative is failing to include a divergence needed by a future execution; this leads to predicate failure and transparent recovery. Performance is slightly slower than without memoization, but observable behavior is identical. Incremental refinement learns from such failures. A false positive includes a divergence that is never needed. This leads to larger and more complicated slices, but has little effect on performance.

To support refinement, ShortCut initially generates a *divergence log* for the reference execution. A Pintool dynamically instruments the replayed execution and generates a compressed log of all program inputs, relative memory accesses, and branches and jumps that can change control flow.

To refine the slice based on an additional recorded execution, ShortCut replays that execution and generates its divergence log. It compares the new log with that of the reference execution and identifies any differences in inputs, data flow, or control flow. It uniquely identifies an instruction that caused a divergence by the instruction address, a per-thread identifier, number of system calls made by that thread since the beginning of the region, and number of basic blocks executed by that thread since the last system call. The latter two fields are specified for the reference execution (e.g., since the number of basic blocks may change for a given instruction if two executions follow different control flow paths).

Refinement records these divergences in a file that is used as input by the slice generator. For inputs, it specifies the bytes that differed. For data flow, it specifies the set of effective addresses accessed by the relative load or store. For control flow, it specifies the diverging instruction, the merge point (where the paths join), and all basic blocks along each observed path. If a path is too complex (i.e., if it contains a system call, synchronization instruction, or more than 512 basic blocks), refinement does not specify the divergence. Optionally, it can automatically end the memoized region when a too-complex control flow divergence occurs.

The final step canonicalizes the divergences. Based on common observations across several executions, ShortCut may anticipate future divergences that are similar to the ones it has actually observed. For data flow divergences, if all accesses fall into a tight range of addresses, canonicalization specifies the anticipated divergence as a range bounded by the minimum and maximum address observed. For control flow divergences, if a given static branch or jump instruction causes divergence at more than one point in the execution, canonicalization asserts that *any* instance where that instruction has tainted inputs may diverge.

Canonicalization has a fundamental tension. Assuming fewer divergences generates smaller slices that run faster but have more predicate failures. Assuming more divergences generates larger and slower slices that fail less often. Our heuristics that resolve this tension affect performance but not correctness since self-verifying slices tolerate both false positives and false negatives.

ShortCut retains the replay log diff for any slice execution that led to transparent recovery. It processes these in batch when spare CPU cycles are available. The current slice is used until offline refinement completes. After creating a new list of canonicalized divergences, it re-runs the slice generator. Finally, it discards the replay and divergence logs for any additional executions since it only needs to retain the much smaller list of actual divergences.

Initially, we tried to generate slices by gathering a large set of profiled runs. We found this worked poorly because naturally-occurring executions have a great deal of *internal non-determinism*, e.g., different thread schedules or different timing for reading asynchronous input. This non-determinism made it difficult to compare two executions. Incremental refinement largely eliminates this internal non-determinism since the slice is gently guided to use the same thread schedule as the reference execution, read asynchronous input in the same-sized chunks, etc.

Still, ShortCut can tolerate some variance. For example, a synthetic execution is only guaranteed to be correct up to the failure point. Executing after the failure point is a form of mutable replay [56]. Since we have not directly observed inputs past this point in slice execution, we are forced to use values from the reference execution (or inferred values consistent with what we saw earlier). After the failure point, an execution using such values may be one that could not occur.

ShortCut could simply use only the portion of the execution prior to failure for profiling. Yet, the code regions after failure often contain useful information with additional data and control flow divergences (divergences are often correlated). Rather than throw this valuable data away, ShortCut uses it because its self-verifying slices tolerate both false negatives and false positives. Incremental refinement continues to use a synthetic execution past the failure point until it reaches a too-complex control flow divergence.

4 Limitations

Experiences with our sample applications have revealed patterns that can prevent ShortCut from generating an effective slice. The X server sends all replies and window events over the same socket. The order of message arrival is occasionally non-deterministic. Slices automatically detect X message reordering and compensate by delaying out-of-order message arrival. While we verified that reordering is correct per the X protocol [43], it is not correct for every protocol. Thus, supporting reordering requires some manual intervention to determine correctness.

Different memory allocation request sizes across executions can lead to different pointer values, which causes ShortCut to include more instructions in the slice. The increase in slice size tends to be limited because of natural padding by the allocator (e.g., to implement a buddy system and increase cache alignment) and because differing pointer values affect data flow more than control flow. To make allocations more consistent, we are currently adding an optimization that pads allocations based on the maximum size seen in previous executions.

Similarly, ShortCut must terminate memoization when a program starts to receive asynchronous, non-deterministic inputs such as unexpected incoming requests or UI events, due to our restrictions on complex control flow divergences. The initial purpose of these restrictions is to simplify the ShortCut code, but this may inherently limit region slice size for some applications. For example, we were trying to accelerate log processing pipelines running as serverless computation [8, 9, 25], but soon discovered that for a realistic benchmark, the input logs varies significantly and thus resulted in complicated control flow divergences that could not be handled under our current restrictions. With more effort, we could handle more complex divergences, but there is some practical limit to complexity that we would reach.

Slice generation and refinement is several orders of magnitude slower than normal execution due to the overhead of dynamic instrumentation. We ameliorate this cost by performing dynamic analysis offline with deterministic record and replay. For background tasks, this cost implies that we should memoize a computation only if we expect it to be repeated several hundred times in the future. For user-facing tasks such as memoizing application startup, this ratio can be much lower since we are using computer time to save the much more valuable user time. Optimizing slice generation will help both use cases; a promising avenue is parallel information flow tracking [46].

ShortCut introduces the potential for information leakage through new side channels. For instance, the slice itself reveals information about the reference execution and subsequent executions. Such channels may make it infeasible to share slices across trust domains, e.g, among different users. ShortCut slices currently use the same ASLR layout as the reference execution. We plan to have the slice take the (random) memory layout as an input parameter and issue relative reads/writes based on the information, which would allow each slice execution to have a different memory layout.

5 Evaluation

Our evaluation answers the following questions:

- How much does ShortCut accelerate applications with mostly-deterministic code regions?
- What is the cost of recovery if predicate verification fails during slice execution?
- How often must predictions be correct?

5.1 Experimental Setup

Web experiments were run with a 4-core server VM with 4 GB of memory, running nginx web server 1.4.7 with PHP 7.0.31. Two identical client VMs run Apache Bench (ab) against the Web server, loading 20,000 pages. There are 8 client threads in total, while the server has 4 threads. All VMs were hosted on a machine with 2 16-core Xeon E5-2687W processors and 256 GB of DRAM. All other experiments were run directly (without virtualization) on a workstation with a 4-core Xeon E5620 processor and 6 GB of DRAM. We ran each experiment 10 times. We report mean values with 95% confidence intervals.

We verified the correctness of all slices executed through byte-by-byte comparison of the resulting address space and registers. To verify a slice, we execute it and checkpoint application memory and registers. Then, we apply the diff of non-deterministic operations (see Section 3.3) to the replay log of the reference execution, replay that log to generate a full (non-memoized) execution with the same inputs seen by the slice, and checkpoint the resulting address space and registers. Comparing checkpoints showed no differences in values, meaning that the memoized and non-memoized executions produced identical results. We verified that system calls executed were equivalent. Finally, we verified that applications behaved as normal after both slice execution and transparent recovery.

5.2 Identifying regions for partial memoization

Our current process for identifying regions to memoize requires us to manually specify the start and end of the sliced region. In the future, we hope to create an automated tool to select regions automatically. For this paper, however, we used the following process to identify slices regions:

First, we identified where each slice should begin. For most benchmarks, this was simply the start of program execution. For the Web/PHP benchmarks, the slice starts when the `php_execute_script` function is called (i.e., the most intuitive point to start a region when we hope to memoize PHP execution).

Next, we set the end of the slice to the point where the control flow diverged beyond what ShortCut allowed (e.g., the divergent path contains a system call or too many basic blocks), i.e., we selected the maximum slice size that worked for the entire profile set. This is a heuristic. As our results will indicate, longer slices that work for most but not all entries in the profile set might be better; shorter slices that are more general could also work well. However, the results do show that this simple heuristic produces reasonable slices.

5.3 Web/PHP server workloads

We use two workloads to measure how effectively ShortCut accelerates PHP-based dynamic content generation in Web servers. This code region is mostly deterministic since it generates similar content for similar requests, and thus can benefit from ShortCut. The first accelerates the generation of the HotCRP user index page that displays conference name, papers submitted, etc. Our disjoint test and training sets change inputs that include the author name, email address, author, paper id, paper title, decision, and PDF link; the training set had 4 Web pages, and the test set had 84. Other system inputs such as the current time vary naturally.

For this benchmark, we found that our initial profile set did not produce a good slice: the slice was too short and the time saved by running a slice was outweighed by fixed performance costs, e.g., for starting the slice. ShortCut's taint tracking tool can identify the specific inputs that led to a divergence. Thus, we looked at the first divergence for the slice, identified which input caused the divergence, and created different profile sets for each value of this input. This tool identified the paper count as the key input, so we divided our original profile set into n profile sets, each with the same number of papers.

Intuitively, the number of papers has a much larger effect on HotCRP control flow than the other parameters, and ShortCut currently tolerates a lot more divergence in data flow than it tolerates in control flow. For instance, if a loop takes a different number of iterations and includes a system call, then ShortCut cannot currently handle that divergence.

The second workload accelerates the generation of MediaWiki pages with disjoint test and training sets that vary the logged-in user, as well as system variables such as the current time; the training set had 2 Web pages and the training set had 89. Since Wiki content is mostly static, MediaWiki has a custom cache for popular pages; however, the PHP scripts that access this cache introduce computational overhead that ShortCut accelerates. The benchmark is short enough that garbage collection is not triggered. Our reported baseline results use the custom content cache, while the slices generated by ShortCut effectively replace such caching.

	HotCRP		MediaWiki	
	baseline	ShortCut	baseline	ShortCut
Throughput (requests/sec)	111.4	437.2	38.9	191.8
Median latency (ms)	70	17	204	41
99% tail latency (ms)	92	40	236	63

Table 1. Throughput and latency for Web workloads

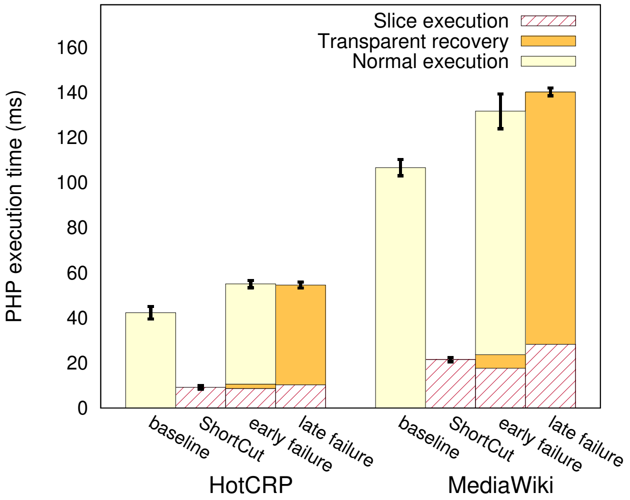

Figure 3. **Speedup for Web/PHP.** Comparing the first two bars in each dataset shows the performance benefit of ShortCut memoization. The last two bars show performance when a predicate fails.

As shown in Table 1, ShortCut speeds up Web server throughput by 392% and 493% for HotCRP and MediaWiki, respectively (as measured by ab). Median latency improves by over a factor of 4 in both cases, and 99% tail latency is improved by 230% for HotCRP and 375% for MediaWiki.

To show how this speedup is achieved, Figure 3 compares the time to execute the memoized region, which roughly corresponds to PHP engine execution. Comparing the first two columns shows that ShortCut partial memoization speeds up the region by 459% for HotCRP and 496% for MediaWiki. The next two columns show PHP execution time when we artificially cause the first and last predicate in the slice to fail (we refer to these scenarios as *early failure* and *late failure*, respectively). With early failure, little work is rolled back during transparent recovery. The cost of late failure is often higher because more work is discarded (but not always due to prefetching benefits of slice execution).

Failing early increases average latency by 30% for HotCRP and 24% for MediaWiki. Failing late increases average latency by 29% for HotCRP and 31% for MediaWiki. As the shadings within the bars show, early failure leads to considerable time re-executing the remainder of the region after recovery, while

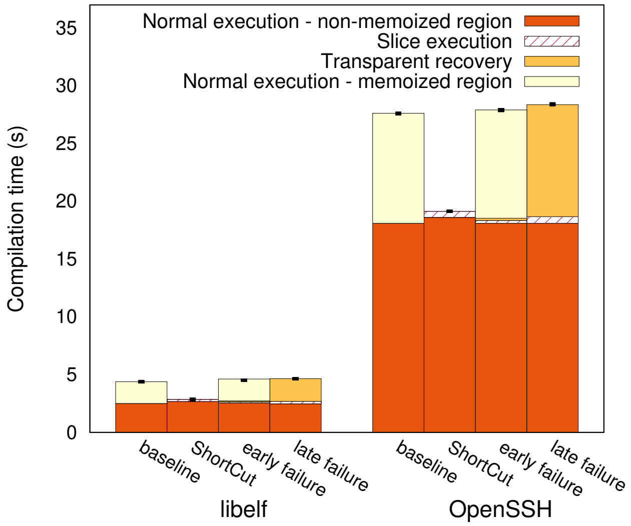

Figure 4. **Speedup for compilation benchmark.** Comparing the first two bars in each dataset shows the performance benefit of ShortCut memoization. The last two bars show performance when a predicate fails.

late failure leads to more time spent in transparent recovery. The reason that early failure does not reduce slice execution time very much compared to successful memoization for this application is that applying the diff dominates total slice execution time, and this work is always done before the first predicate is verified. The diff could potentially be applied incrementally as the slice runs, which would be a nice future optimization for early failure.

Since the performance impact of predicate failure is much less than the improvement seen from predicate success, predicates need not succeed very often to improve average performance. ShortCut breaks even on average performance if predicates succeed only 23–28% of the time for the two scenarios under both the early and late failure scenarios. If predicates succeed more often than this, ShortCut improves average performance. This is a reasonable expectation based on our experiments, since predicates succeed for all 20,000 requests in the throughput test.

For server workloads, we can also consider another break-even point: how many page loads are needed to offset the computational time to generate the slice? For HotCRP, the index page must be loaded 2809 times to break even. For MediaWiki, a page must be loaded 2236 times to break even. These results are encouraging: even moderately popular pages exceed these values.

5.4 Compilation

Developers spend considerable time waiting for code to compile, especially when iteratively making many small changes and testing. Make files, ccache [14], etc. elide work when all inputs to a compilation are *exactly* the same, but not when inputs are only mostly similar.

We use ShortCut to memoize gcc compilation of the libelf-0.8.9 and openssh-6.1p1 libraries. Our training sets make various changes to source files throughout the C files so that the memoized region for each file ends shortly after the headers are compiled; this results in a slice for each C file. These memoized code regions are mostly deterministic and thus can benefit from ShortCut. Our test set executes a find-and-replace on a variable name that causes recompilation of 34 out of 53 C files in libelf and 136 of 198 C files in openssh.

Figure 4 compares the time to rebuild the library packages with and without ShortCut (this time includes compilation, linking, and all other steps required to build each library). Memoization speeds up libelf build by 54% and openssh build by 43%. The memoized regions execute 10–18 times faster than the non-memoized regions, so benchmark performance is mostly gated by the ratio of computation used to compile the initial header block to that used for the remainder of each file.

The last two bars in each dataset in Figure 4 show build time when the first and last predicate fails for all recompiled files. Early failure leads to a 1–3% slowdown compared to unmemoized execution, while late failure causes a 3–6% slowdown. Because the performance benefit of memoization is high and the cost of predicate failure is low, only an average 5–8% of compilations need to see all predicates succeed for ShortCut to show performance benefits for early failure, and 10–14% must succeed to reach the break-even point for late failures.

Finally, we compare ShortCut to a custom optimization, gcc's precompiled headers [24]. Use of precompiled headers only speeds up package build by 20% for libelf and 26% for openssh. In other words, ShortCut achieves an additional 27% and 13% speedup compared to this hand-crafted optimization, respectively. One reason that ShortCut outperforms precompiled headers is that "only one precompiled header can be used in a particular compilation [24]"; this is a good example of how the complexity of implementing optimizations can limit their effectiveness. On the other hand, precompiled headers can be more general (e.g., for libraries used by multiple applications), and they are faster to generate than ShortCut slices.

5.5 Interactive application startup

An application often behaves similarly each time it starts; e.g., it reads the same configuration and interacts in standard ways with the OS to acquire resources. Therefore, application startups are good candidates for ShortCut to accelerate. We consider 4 interactive applications, chosen for variety:

- **GNU Emacs**: Emacs startup is slow when a file has complex formatting. Our benchmark invokes Emacs to repeatedly edit a complex 1.3 MB HTML file (a Yahoo baseball statistics page). Our disjoint test and training sets modify pitcher data near the end of the file (in the last 10% of bytes).

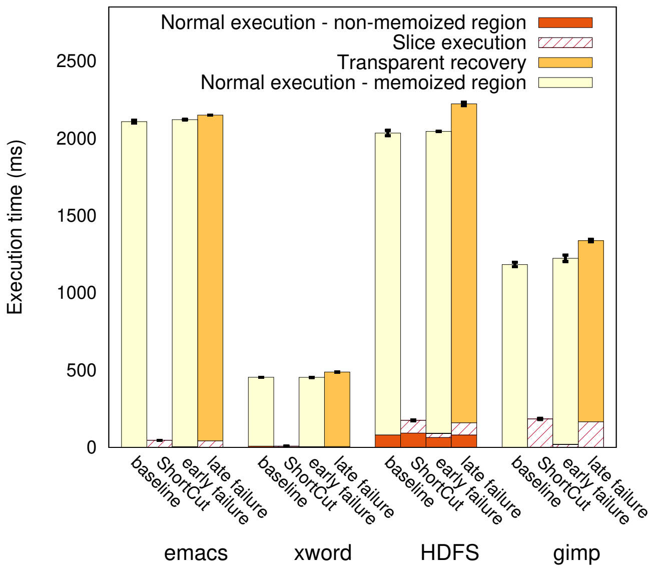

Figure 5. **Speedup for application startup.** The top segment of each bar shows the time to execute the startup region; for xword and HDFS, the bottom segments shows the time to finish running the application to completion.

- **GIMP**: Complex GUI applications like GIMP are slow to start due to reading configuration data, opening X windows, etc. Our disjoint test and training sets start GIMP repeatedly from the launch menu.
- **HDFS**: Java startup is slowed by loading class files. We run the benchmark of Lion et al. [37], in which the HDFS file utility is used to read file contents. Our disjoint test and training sets read different files approximately 10 MB in size from HDFS.
- **xword**: Custom data pipelines used by small communities often include minimally-optimized tools. As an example, we use a crossword generator developed by an author and used for commercial crossword construction. This constraint solver finds a maximum-value placement of words from a large list in a specified grid of black and white squares. Startup is slow because the tool initializes complex data structures. Typically, hundreds or thousands of grid patterns are tried to find the best solution and many grids are quickly discarded. Our disjoint test and training sets use different patterns grids (which change frequently) and a common wordlist (which changes infrequently).

For HDFS and xword, we report the time for the application to complete. For Emacs and GIMP, we report the time until the user can interact with the application.

Figure 5 shows ShortCut speeds up startup of the 4 applications by a factor of 6–76; the average startup is over 27 times faster. GIMP shows the least acceleration because the memoized region interacts with external processes such as the X server and the window manager. While the region is accelerated, these other processes are not, so the slice often waits for these other processes to respond. Xword achieves the greatest acceleration because it makes relatively few system calls. ShortCut speeds up user-level computation but not kernel computation since the slice makes essentially the same system calls as the non-memoized region.

Failure and transparent recovery is not substantially slower than normal startup. The average increase in startup time after an early failure is only 1%; the average increase after a late failure is 7%. GIMP has the largest overhead because its slice takes longest to execute; if it fails late, substantial work is redone. The break-even point (i.e., how often all predicates must pass for ShortCut to improve startup performance) is only 3% for Emacs, 6% for xword, 9% for HDFS, and 22% for GIMP.

5.6 Application and slice characteristics

Table 2 provides detailed characteristics of each slice and memoized region. For compilation, we aggregate results for all memoized C files. The first two rows in the table show the size of the memoized region in the reference execution (i.e., the number of dynamic x86 instructions executed). These large regions range from hundreds of millions to billions of instructions. The generated slices are much smaller, executing on average 0.2% of the user-level x86 instructions.

The next two rows shows the wall-clock time required to execute the memoized region of the reference execution and the time required to generate the corresponding slice. Slice generation generally takes several minutes.

Slices verify a large number of predicates during execution, with input predicates being the most common, partially because they are verified on a per-byte basis. The applications exhibit considerable variance in the number of index, control flow, and input divergences supported by the slice. Although some applications like GIMP have many control flow predicates to verify that the slice execution follows the same general control flow as the reference execution, there are few actual control flow divergences in the sliced regions. On average, only 2% of the bytes in a diff are modified by a slice; yet, such modifications are typically several megabytes in size. As a result, none of our benchmarks produce correct results with generic memoization.

6 Related Work

ShortCut builds on prior work in four related areas: memoization, incremental computation, program slicing, and replay recovery. Memoization stores the results of a repeated computation and returns the stored result in lieu of performing the same computation. iThreads [10] supports parallel incremental computation by memoizing sub-computations between synchronization calls. Nectar [26] memoizes deterministic LINQ programs and enables incremental computation in the data center. One use of lightweight contexts [38] is memoizing generic PHP startup. Memoization has also been applied at the level of individual functions [27, 30] and new programming models have been proposed to support

	Web/PHP		Compilation		Interactive application startup			
	HotCRP	MediaWiki	libelf	OpenSSH	Emacs	xword	HDFS	GIMP
Region instructions (millions)	141	379	4547	23166	7440	1156	4354	2951
Slice instructions (millions)	0.33	0.72	7	42	19	2	11	29
Reference execution time (s)	0.042	0.11	4.4	27.5	2.1	0.45	2	1.2
Slice generation time (s)	118	246	775	1655	584	94	461	665
Index predicates	573	1014	4594	745	808731	2894	193871	161829
Control flow predicates	2802	25016	126	22517	1346457	209077	900754	2733256
Input predicates (bytes)	120284	728594	3790436	20554303	380699	130084	2051889	1514187
Index divergences	216	329	0	0	84	1324	9	628
Control flow divergences	23	33	0	0	280	401	824	0
Input divergences (bytes)	55082	222918	579792	5544406	75450	1497	404619	229488
Diff size (MB)	4.4	10.9	285	1186	25	680	814	225
Number of slices	1	1	68	272	1	1	21	13
Slice size (MB)	0.47	1.6	12	14	55	5	31	95
Locations modified (MB)	0.6	0.7	3.3	15.6	0.3	6.3	5.4	9.7

Table 2. **Detailed statistics for each application.**

memoization [39, 52, 54]. These prior results all require per-region input equality and determinism; i.e., the memoized computation must be the same for a region each time. In contrast, ShortCut operates on mostly-deterministic regions that differ across executions, and this lets ShortCut's regions scale to billions of dynamic instructions.

Adaptive and incremental computation are well-studied topics in the programming language community. Like ShortCut, adaptive programming can provide a partial memoization capability by only re-executing the portion of computation affected by changed inputs. However, these are not generic, systems-level approaches: they either propose new programming languages, models, or libraries [1, 2, 12, 13, 29, 35] and/or work only for functional programming languages [15, 45]. Program specialization is a program transformation technique that generates a specific implementation for a program fragment dedicated to a known context or input. It has been used for both functional [5, 17] and imperative languages [6, 16, 49]. Program specialization usually requires source code and may also require users to describe specialization opportunities via a declarative language or specify invariants in the source code [16]. ShortCut can be viewed as a form of adaptive computation or program specialization, but it can potentially be applied to any executable and operates at instruction level. Its combination of slicing and checkpointing avoids the need for source code changes and reduces the burden of users, and, vitally, predication and transparent recovery make slice generation tractable and greatly reduce slice size. ShortCut partial memoization scales to unprecedented region size compared to all of these prior approaches.

Delta execution, which deduplicates similar computation, has been used for server validation and auditing [32, 33, 51, 55]. One can regard ShortCut as deduplicating a reference computation with unknown, future computations, whereas delta execution deduplicates known, past computations.

There have been many efforts to accelerate application startup in Linux [31, 36], Windows [41, 50], Windows phones [58], and Android [34, 44]. These approaches do not eliminate user-level computation, but they provide complementary benefits such as ensuring that binaries and libraries remain in memory or are loaded quickly. Other efforts accelerate startup through application-specific approaches such as JVM Warm-Up [37] or gcc's precompiled headers [24]. Application-specific efforts can have similar benefits as ShortCut but introduce considerable complexity; e.g., precompiled headers have many restrictions such as using only one such header per compilation written in the same language as source and having no C token preceding the header. In contrast, ShortCut's generic support requires no application modification or source code changes.

Program slicing is a classic technique [3, 53, 57] that has been applied to fault localization [4, 28] and generating likely invariants [48]. ShortCut proposes checkpoint modification as a novel application, and it generates usable slices over very large and complex code regions. Rollback recovery [23] and deterministic replay [11, 21] are other classic techniques used by ShortCut. ShortCut builds on Arnold [20] replay but differs from this prior work by introducing the notion of synthetic executions.

7 Conclusion

ShortCut improves application performance through partial memoization of mostly-deterministic code regions. It modifies a generic diff with the actual values seen during subsequent executions. Predication makes slicing regions with billions of dynamic instructions feasible, and rollback recovery via synthetic executions makes predication correct.

Acknowledgments

We thank the anonymous reviewers and our shepherd Taesoo Kim for their thoughtful comments. This work has been supported by the National Science Foundation grant CNS-1513718 and by a gift from NetApp.

References

[1] Umut A. Acar. *Self-adjusting Computation*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2005.

[2] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. *ACM Trans. Program. Lang. Syst.*, 28(6):990–1034, November 2006.

[3] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 246–256, 1990.

[4] Hiralal Agrawal, Joseph R. Horgan, Saul London, and W. Eric Wong. Fault localization using execution slices and dataflow tests. In *Proceedings of the 6th International Symposium on Software Reliability Engineering*, October 1995.

[5] María Alpuente, Moreno Falaschi, and Germán Vidal. Partial evaluation of functional logic programs. *ACM Trans. Program. Lang. Syst.*, 20(4):768–844, July 1998.

[6] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[7] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation*, Hollywood, CA, October 2012.

[8] AWS Lambda. https://aws.amazon.com/lambda/.

[9] Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[10] Pramod Bhatotia, Pedro Fonseca, Umut A. Acar, Bjorn Brandenburg, and Rodrigo Rodrigues. iThreads: A threading library for parallel incremental computation. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, Istanbul, Turkey, March 2015.

[11] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault-tolerance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 1–11, Copper Mountain, CO, December 1995.

[12] Sebastian Burckhardt, Daan Leijen, Caitlin Sadowski, Jaeheon Yi, and Thomas Ball. Two for the price of one: A model for parallel and incremental computation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 427–444, 2011.

[13] Magnus Carlsson. Monads for incremental computing. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, pages 26–35, 2002.

[14] ccache — a fast C/C++ compiler cache. https://ccache.dev/documentation.html.

[15] Yan Chen, Joshua Dunfield, Matthew A. Hammer, and Umut A. Acar. Implicit self-adjusting computation for purely functional programs. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, pages 129–141, 2011.

[16] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, E.-N. Volanschi, J. Lawall, and J. Noyé. Tempo: Specializing systems applications and beyond. *ACM Comput. Surv.*, 30(3es), September 1998.

[17] Charles Consel. A tour of schism: A partial evaluation system for higher-order applicative languages. In *Proceedings of the 1993 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '93, pages 145–154, New York, NY, USA, 1993. ACM.

[18] Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 337–351, 2011.

[19] David Devecsery, Peter M. Chen, Jason Flinn, and Satish Narayanansamy. Optimistic hybrid analysis: Accelerating dynamic analysis through predicated static analysis. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*, Williamsburg, VA, March 2018.

[20] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. Eidetic systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation*, Broomfield, CO, October 2014.

[21] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 211–224, Boston, MA, December 2002.

[22] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, September 2002.

[23] E. N. Elnozahy and Willy Zwaenepoel. Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit. *IEEE Transactions on Computers*, C-41(5):526–531, May 1992.

[24] Using the GNU compiler collection (gcc). http://gcc.gnu.org/onlinedocs/gcc/Precompiled-Headers.html.

[25] Google Cloud Functions. https://cloud.google.com/functions/.

[26] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. Nectar: Automatic management of data and computation in datacenters. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, Vancouver, BC, October 2010.

[27] Philip J. Guo and Dawson Engler. Using automatic persistent memoization to facilitate data analysis scripting. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA)*, pages 287–297, New York, NY, USA, 2011.

[28] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. Locating faulty code using failure-inducing chops. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, November 2005.

[29] Matthew A. Hammer, Khoo Yit Phang, Michael Hicks, and Jeffrey S. Foster. Adapton: Composable, demand-driven incremental computation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 156–166, 2014.

[30] Allan Heydon, Roy Levin, and Yuan Yu. Caching function calls using precise dependencies. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, 2000.

[31] Yongsoo Joo, Junhee Ryu, Sangsoo Park, and Kang G. Shin. Fast: Quick application launch on solid-state drives. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, 2011.

[32] Taesoo Kim, Ramesh Chandra, and Nickolai Zeldovich. Efficient patch-based auditing for Web application vulnerabilities. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation*, Hollywood, CA, October 2012.

[33] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Intrusion recovery using selective re-execution. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, Vancouver, BC, October 2010.

[34] Joohyun Lee, Kyunghan Lee, Euijin Jeong, Jaemin Jo, and Ness B. Shroff. Context-aware application scheduling in mobile systems: What will users do and not do next? In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 1235–1246, 2016.

[35] Ruy Ley-Wild, Matthew Fluet, and Umut A. Acar. Compiling self-adjusting programs with continuations. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, pages 321–334, 2008.

[36] Linux application preloading. https://wiki.archlinux.org/index.php/Preload.

[37] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. Don't get caught in the cold, warm-up your JVM: Understand and eliminate JVM warm-up overhead in data-parallel systems. In *Proceedings of the 12th Symposium on Operating Systems*

Design and Implementation, November 2016.

[38] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-weight contexts: An OS abstraction for safety and performace. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation*, November 2016.

[39] Dionysios Logothetis, Christopher Olston, Benjamin Reed, Kevin C. Webb, and Ken Yocum. Stateful bulk processing for incremental analytics. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 51–62, 2010.

[40] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 190–200, Chicago, IL, June 2005.

[41] Jeanna Matthews, Sanjeev Trika, Debra Hensgen, Rick Coulson, and Knut Grimsrud. Intel® turbo memory: Nonvolatile disk caches in the storage hierarchy of mainstream computer systems. *ACM Transactions on Storage*, 4(2):4:1–4:24, May 2008.

[42] James Newsome and Dawn Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, February 2005.

[43] Adrian Nye, editor. *X Protocol Reference Manual*. O'Reilly and Associates, Inc., 1990.

[44] Abhinav Parate, Matthias Böhmer, David Chu, Deepak Ganesan, and Benjamin M. Marlin. Practical prediction and prefetch for faster access to applications on mobile phones. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 275–284, 2013.

[45] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 315–328, 1989.

[46] Andrew Quinn, David Devecsery, Peter M. Chen, and Jason Flinn. JetStream: Cluster-scale parallelization of information flow queries. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation*, Savannah, GA, November 2016.

[47] Michiel Ronsse and Koen De Bosschere. RecPlay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2):133–152, May 1999.

[48] Swarup Kumar Sahoo, John Criswell, Chase Geigle, and Vikram Adve. Using likely invariants for automated software fault localization. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, Houston, TX, March 2013.

[49] Ulrik P. Schultz, Julia L. Lawall, and Charles Consel. Automatic program specialization for java. *ACM Trans. Program. Lang. Syst.*, 25(4):452–499, July 2003.

[50] Performance-enhancing features on Windows Vista. https://technet.microsoft.com/en-us/library/2007.03.vistakernel.aspx.

[51] Cheng Tan, Lingfan Yu, Joshua B. Leners, and Michael Walfish. The efficient server audit problem, deduplicated re-execution, and the web. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, pages 546–564, 2017.

[52] Yang Tang and Junfeng Yang. Secure deduplication of general computations. In *Proceedings of the 2015 USENIX Conference on USENIX Annual Technical Conference*, 2015.

[53] Frank Tip. A survey of program slicing techniques. Technical report, Centre for Mathematics and Computer Science, 1994.

[54] Hung-Wei Tseng and Dean M Tullsen. Data-triggered threads: Eliminating redundant computation. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 181–192, 2011.

[55] Joseph Tucek, Weiwei Xiong, and Yuanyuan Zhou. Efficient online validation with delta execution. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 193–204, 2009.

[56] Nicolas Viennot, Siddharth Nair, and Jason Nieh. Transparent mutable replay for multicore debugging and patch validation. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2013.

[57] Mark Weiser. Program slicing. *Proceedings of the 5th International Conference on Software Engineering (ICSE)*, pages 439–449, March 1981.

[58] Tingxin Yan, David Chu, Deepak Ganesan, Aman Kansal, and Jie Liu. Fast app launching for mobile devices using predictive user context. In *Proceedings of the 10th International Conference on Mobile Systems, Applications and Services*, pages 113–126, Lake District, United Kingdom, June 2012.

[59] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 1–14, Banff, Canada, October 2001.