SpanCoverage: Architectural Support for Increasing the Path Coverage of Dynamic Bug Detection

Shan Lu, Pin Zhou, Wei Liu, Yuanyuan Zhou and Josep Torrellas
Department of Computer Science,
University of Illinois at Urbana Champaign

Abstract

Dynamic tools for software bug detection such as Purify are commonly used because they leverage run-time information. However, they suffer from a fundamental limitation, namely the Path Coverage Problem: they detect bugs only in taken paths but not in non-taken paths. In other words, they require bugs to be exposed in the monitored execution.

This paper makes one of the first attempts to address this fundamental problem with a simple architecture extension. We propose SpanCoverage, a novel design that dynamically increases the path coverage of dynamic bug detection tools with no involvement from programmers. As a program executes, SpanCoverage also selectively executes non-taken paths in a hardware sandbox without side effects. This enables dynamic bug detection tools to find bugs in these paths that would otherwise not be detected. To minimize overhead, SpanCoverage provides an optimization option to leverage thread level speculation (TLS) with small modifications to execute selected non-taken paths on idle processors in a chip multi-processor (CMP) architecture.

We evaluate SpanCoverage using three dynamic bug detection methods: assertions, software-only checkers (CCured), and hardware-assisted checkers (iWatcher). Our experiments with nine buggy programs using inputs that do not expose the tested bugs show that SpanCoverage is able to help these tools detect seven out of the nine tested bugs that are otherwise not detected. This is because SpanCoverage increases the program branch coverage of each input from 36% to 60% on average. The cumulative coverage also improves significantly by 11–58%, when applications are tested with multiple input sets. We also show that SpanCoverage has a modest overhead (less than 9.9% for three open source applications and three SPEC2000 benchmarks with the CMP optimization) and introduces only a few (3 on average) false positives.

Key Words: architectural support, software bugs, dynamic bug detection, path coverage, multi-path execution

1 Introduction

1.1 Motivation: The Path Coverage Limitation of Dynamic Bug Detection

Software bugs are one of the major causes of system down time and security attacks. Specifically, software bugs cost the U.S. economy $59.5 billion annually (0.6% of the GDP) [28] and cause about 40% of computer system failures [25]. Although many debugging tools have been proposed to detect bugs automatically, the ubiquity of software bugs is a strong testimony to the fact that more innovations in this topic are needed.

Dynamic bug detection [4, 7, 13, 15, 49, 50] is a commonly used method in detecting software bugs. These tools detect bugs by monitoring programs’ execution at run-time. Therefore, they have more accurate information on
variable values and aliasing information. As a result, they can catch more bugs and report fewer false positives than static checkers. Examples of this method include assertions, software-only dynamic checkers such as Purify [15], DIDUCE [13] and CCured [7], and hardware-assisted dynamic checkers such as iWatcher [50] and AccMon [49].

Unfortunately, almost all dynamic bug detection tools suffer from a major limitation: the path coverage problem, i.e., they can only detect those bugs which appear in the executed paths [10]. In other words, they require the bug to manifest itself during the monitored runs! If a bug is present in a non-taken path (a control path which is not executed) in a monitored run, dynamic tools will fail to detect it. This limitation has been commonly pointed out by previous static checkers as a weakness of dynamic bug detection tools [10].

Before we use several examples to demonstrate this limitation, we first explain some terminology. In this paper, we use a branch edge to denote one of the two edges (true and false) after a branch instruction, and a path to denote a sequence of multiple consecutive branch edges. We use branch coverage (also known as link coverage or decision coverage), a commonly used testing coverage metric in software testing, to measure the percentage of the tested program’s branch edges that are executed in the monitored run and thereby are checked by the underlying dynamic bug detection tool.

Figure 1: Another bug example from Siemens benchmark: bug can be detected only if the path: line 210→103→104→211 is executed.

Figure 2: A general example where the branch condition is related to the system states, hardware states, resource states and/or application states.

Figure 1 gives a simple example to illustrate the path coverage problem. This code segment comes from the schedule benchmark in the Siemens Benchmark Suite, a commonly used benchmark suite for software testing. Line 211 contains a null-pointer dereference bug: if variable list is NULL, find will return NULL and so line 211 will dereference a NULL pointer, causing access violations. However, if the path (line 210→103→104→211) is not executed, elem will not be NULL. As a result, this bug will not manifest and thus cannot be detected by the underlying dynamic bug detection tool in the monitored run. This bug, thereby, escapes into production runs, causing software to fail or being exploited by malicious users to launch denial-of-service attacks. This example requires a path, a sequence of 3 branch edges, to be executed in order to expose the bug.

As shown in Figure 2, a more general and complex example is when a bug occurs in a control path which is taken only under a combination of certain system states (e.g., the time of day), hardware states (e.g., disk states), resource states (e.g., how much virtual memory has been allocated), as well as application states (e.g., the total number of outstanding requests). In order to check such a path using a dynamic bug detection tool, besides providing appropriate input values, programmers also need to generate necessary combinations of states.
1.2 Why Software Testing is Not Enough?

One might wonder why the path coverage is a limitation for dynamic bug detection tools, and why programmers cannot simply run as many tests as possible to make sure every path is checked by the dynamic bug detection tool. The reasons are as follows:

- The problem of automatically generating an input to cover a given control path is theoretically uncomputable (the halting problem can be reduced to this problem) [38]. Although many studies [9, 32] have examined how to generate test cases, most of them are based on program specification, which is much more abstract than real implementation. As a result, the generated test suite is still far from providing a complete coverage of all code or branch edges, not to mention all execution paths (sequences of multiple branch edges).

- The problem becomes even harder if the control path also depends on system, hardware, resource and application states as shown in Figure 2. While fault injection-based testing can address this problem to some extent, it is usually very expensive and cannot test all possible state combinations (especially for application states and resource states) and hence all possible paths that depend on these combinations.

- Since a path is a sequence of multiple branch edges, the total number of feasible paths in a program can be very large and even infinite due to the existence of loops and recursive functions. Therefore, it is impossible to test every path.

- Since it is difficult to automatically generate a good test suite with a reasonably good coverage, achieving this goal would require a large amount of human efforts [22]. This is why software companies such as Microsoft usually hire many testing engineers to design and implement test cases.

- Dynamic bug detection tools are usually applied to a very small number of selected test runs because many dynamic tools, such as Purify and Valgrind, incur very large (up to 40-100 times) overheads [49], too time consuming to monitor hundreds or even thousands of test runs, especially for large software. For example, if each test run without Purify takes 2 hours, using Purify to check 100 test runs would require 333-833 days!

Therefore, it is desirable to automatically increase the path coverage for any single monitored run to allow the underlying dynamic bug detection tool to detect more bugs.

1.3 Other Types of Coverage

A recent work conducted by Larsen and Austin addressed a closely related, complementary, but different coverage problem called value coverage (value ranges of variables) for detecting buffer or string overflows [21]. This work increases the value coverage for detecting buffer overruns by shadowing each input value with an interval constraint variable. At potentially dangerous uses of inputs, such as array references, the entire range of an input value is validated using the computed interval constraint. This way, even if the user-specified input value does not directly expose the bug, the dynamic buffer overrun monitoring system can still detect it. Since it is a software-only solution, it slows down applications by 13-220 times.

While the above work is very useful in increasing the value coverage for buffer overflow detection, as also acknowledged by the authors themselves, their technique do not address the path coverage problem. This is because it follows only taken paths instead of exploring both taken and non-taken paths. In the example shown in Figure 1, their technique will not check the buggy path since the monitored executions do not follow this path. In contrast, our work exactly addresses this path coverage problem. Therefore, our work well complements theirs: their work addresses the value coverage problem, while our work addresses the path coverage problem.
1.4 Our Contributions

In this paper, we make one of the first attempts to address the fundamental path coverage problem using a simple architectural extension. Specifically, we propose an innovative, automatic, low-overhead, and general approach, called SpanCoverage, that allows dynamic bug detection tools to detect bugs in non-taken paths (called NTPaths) that would otherwise not be detected. These NT-Paths are executed in a hardware “sandbox” without affecting programs’ normal execution. To minimize overhead, SpanCoverage provides an optimization option to leverage a modified thread level speculation (TLS) to execute NT-Paths on idle cores in a Chip Multiprocessor (CMP) architecture.

The feasibility of our idea is first validated through our crash latency analysis using ten applications (Section 3.2). To address the state inconsistency problem in NT-Path execution that may introduce false positives in bug detection, SpanCoverage uses a compiler to insert predicated instructions to fix key variables (Section 4.4).

The main characteristics that distinguish SpanCoverage are:

- **Generality.** SpanCoverage makes no assumption about bug types or dynamic bug detection methods (tools). Therefore, it can increase the path coverage of any dynamic bug detection tools in detecting various types of bugs such as invariant violations, accesses to freed memory, buffer overflows, semantic bugs, etc.

  We evaluate SpanCoverage using three different dynamic bug detection methods: (1) assertions; (2) a software-only dynamic checker, CCured [7]; and (3) a hardware-assisted dynamic checker, iWatcher [50]. Our experiments with nine buggy programs using inputs that do not expose the tested bugs show that SpanCoverage is able to help these tools detect seven out of nine tested bugs, which are not detected by them otherwise (without SpanCoverage) due to their path coverage limitation.

- **Improving Path Coverage.** SpanCoverage significantly increases the branch coverage of the monitored execution from 36% to 60% on average. Even when multiple test inputs are used for each application, the cumulative branch coverage improvement by SpanCoverage is still significant (by 11–58%).

- **Reducing Human Efforts.** SpanCoverage minimizes the large amount of efforts from software engineers in designing and implementing test cases to check those non-taken paths that are automatically covered by SpanCoverage.

- **Low Overhead.** With the standard configuration (no CMP optimization), SpanCoverage imposes 59.9–142.3% overhead for 3 real applications, and 4.5–23.2% for the 3 tested SPEC2000 benchmarks, which is definitely tolerable compared to the 40X–100X slowdowns imposed by many dynamic bug detection tools. With the CMP optimization, the overhead is further reduced to less than 9.9%, which indicates that SpanCoverage can be potentially used during production runs.

- **Few False-Alarms.** Because SpanCoverage uses predicated instructions to fix key variable values before the execution of an NT-Path, it significantly reduces the number of false positives in bug detection to only a few (3 on average) for our seven tested buggy applications.

This paper is organized as follows. Section 2 describes some background. Sections 3 and 4 describe the idea and architecture of SpanCoverage. Sections 5 and 6 present the evaluation methodology and experimental results. Section 7 discusses related work. Section 8 concludes the paper.
2 Background

2.1 Testing Coverage

Testing coverage measures how much a program is executed in a monitored run. It directly affects the effectiveness of dynamic bug detection tools: the higher the coverage, the more bugs can potentially be detected. There are various ways to measure the testing coverage. The simplest one is statement coverage, i.e., the percentage of executable statements executed in the monitored runs [30]. It is easy to measure but has a major shortcoming: insensitive to control structures, to which many bugs are related. To address this problem, a better alternative is the branch coverage (also called as decision coverage or link coverage), measuring how many branch edges have been executed in the monitored runs [30]. However, this is still limited in that it only looks at one branch at a time.

One of the most accurate code coverage metrics is path coverage [5]. It focuses on the entire execution path, i.e. the sequence of all branch decisions from the very beginning to the program exit. Path coverage is stronger than statement and branch coverage because, even if a certain statement or branch decision has already been touched, due to different branch evaluation history, some combinations of branch edges may never be tested. Unfortunately, path coverage is hard and in many cases impossible to measure due to the infinite number of paths. Therefore, statement coverage and branch coverages are the dominant testing coverage criterion used in software testing. In our evaluation, we can only show SpanCoverage’s improvement on branch coverages, even though many of our design decisions are targeted for improving path coverages.

High testing coverage is always hard to achieve. Especially since as coverage increases, further coverage improvement becomes increasingly difficult. In large softwares, after around 80-90% branch coverage, each percentage of coverage increase requires a huge amount of effort [38]. Due to this difficulty, good engineering practices usually set 70-90% branch or statement coverage as the target. As a result, bugs located in the remaining 10-30% uncovered program parts will inevitably slip into production runs. As shown in our experimental results, SpanCoverage is able to push for the remaining 10-30% of branch coverage to reach nearly 100% coverage for some applications, and thus enhance software robustness in production runs.

2.2 Branch Mutation

SpanCoverage needs to perform branch mutation (forcing execution of a non-taken path) to increase the path coverage of dynamic bug detection tools. Forcing execution of a non-taken path may result in state inconsistency (some variable values are inconsistent with the path taken), which may lead to abnormal behaviors such as a crash along the execution of this wrong path.

Fortunately, several recent studies [11, 46] on hardware fault injection have examined the effects of random branch mutation and their results show that, in a large percentage (30-40%) of the cases, the state inconsistency caused by injected branch mutations does not manifest at all and programs continue executing correctly without either crashes or silent errors until the end of the program. For example, the Y-branch study [46] conducted by Patel et al. shows that 40% of branch mutations do not affect the correctness of program behavior. Iyer et al.'s study [11] shows that 33% of random branch mutations in the Linux kernel are not manifested at all. One of the reasons for such results is that many of these branches are just optimization paths or short-cuts.

These studies provide a good foundation for our work. To further validate the feasibility of our approach, we have conducted a crash latency analysis using four applications (Section 3.2).
2.3 Thread Level Speculation (TLS)

The standard configuration of SpanCoverage requires only simple checkpoint and rollback extensions to sandbox non-taken paths’ updates. However, to leverage the emerging CMP architecture for reducing overhead, SpanCoverage also provides an optimization option by modifying a thread-level speculation (TLS) architecture to execute non-taken paths concurrently with taken paths. In this subsection, we briefly describe the TLS architecture on which SpanCoverage’s CMP optimization is based. In Section 4.3, we present SpanCoverage’s modification to TLS.

TLS is an architectural technique for speculative parallelization of sequential programs [6, 41, 42, 43]. TLS can be built on a multi-threaded architecture, such as an SMT or CMP architecture. With TLS, the execution of a sequential program is divided into a sequence of threads (also called tasks, slices, or epochs). These threads are then executed speculatively in parallel, while special hardware detects violations of the program’s sequential semantics. Any violation results in the incorrectly executed thread being squashed and re-executed. To enable squashing and re-execution, typically, the memory state of each speculative thread is buffered in caches or special buffers. When a thread finishes its execution and becomes safe, it can commit. Committing a thread merges its state with the safe memory. To guarantee sequential semantics, threads commit in order. In the baseline TLS architecture, each cache line is tagged with a version, i.e. the ID of the thread to which the line belongs. More details about the baseline TLS architecture can be found in [6, 34].

Note that SpanCoverage does not use TLS to parallelize sequential execution. Instead, it uses TLS in a CMP to sandbox the side effects of non-taken paths and keep track of data dependencies, which allows non-taken paths to be executed concurrently to taken paths.

3 SpanCoverage Idea

3.1 Idea Overview

The main purpose of SpanCoverage is to enable dynamic bug detection to check bugs on both taken and non-taken paths, so that potentially more bugs can be detected from a monitored run. SpanCoverage does this by “silently” (without side effects) executing non-taken paths (called as NT-Paths) in a hardware sandbox. To minimize overhead, SpanCoverage provides an optimization option to leverage idle cores (processors) in a CMP architecture.

Figure 3 shows the main idea of SpanCoverage. At a branch $a$, suppose the program follows the right path after the branch condition is resolved. SpanCoverage takes a checkpoint and redirects the execution to the non-taken path.
i.e. the left path. When the left path is executed, its memory updates are sandboxed so that they are invisible to the program’s normal execution.

To implement this idea, the standard configuration of SpanCoverage uses a simple checkpoint-and-rollback support as shown in Figure 4. To reduce overhead, SpanCoverage avoids exploring all non-taken paths. Instead, it selectively executes those non-taken paths that are more likely to have undetected bugs. Intuitively, if a path has been executed as a taken path for many times during the monitored run, it is less likely to have undetected bugs. Therefore, SpanCoverage executes those non-taken paths that so far have not been well exercised in the monitored execution.

To further reduce overhead, SpanCoverage can optionally leverage the CMP architecture by executing an NT-Path on an idle core while the taken path continues executing using the primary core. As shown on Figure 5, after branch a, the non-taken path and the taken path execute concurrently on different cores. At a subsequent branch instruction b, a new non-taken path can be explored in another idle core before the previous one terminates.

Since SpanCoverage targets for improving path coverage, an NT-Path does not necessarily stop at the next branch instruction or at the time its control flow merges with the taken path. In the example shown on Figure 1, if the NT-Path (line 103-104) stops after function find returns, the bug cannot be detected by the underlying dynamic bug detection tool. Therefore, SpanCoverage continues running a NT-Path for a threshold number of instructions unless some events prevent it to do so (See Section 4.2).

3.2 Feasibility Analysis

Following a non-taken path can result in a state inconsistency problem: the condition variables and correlated variables do not match with the path taken. This problem may lead to an immediate crash (e.g. divide-by-zero, access violations, etc) in an NT-Path. Even though SpanCoverage handles crashes in an NT-Path by discarding the exceptions without delivering them to the OS (Section 4.2), too many crashes may prevent NT-Paths from running long enough for the underlying dynamic bug detection tool to detect possible bugs.

Fortunately, as described in Section 2, previous works [46, 11] have shown that 30–40% of branch mutations do not result in any crashes or even silent errors. This condition is stronger than what is required by SpanCoverage because, different from what were experimented in these previous works, SpanCoverage does not execute an NT-Path till the end of the program.

Besides crashes, another reason that may prevent an NT-Path from running long enough to detect bugs is that: the hardware can no longer sandbox some special side effects of an NT-Path. The current sandbox implementation of SpanCoverage cannot roll back side effects made to the OS or the I/O subsystems. Therefore, when any such event occurs during an NT-Path execution, the NT-Path needs to be squashed. To sandbox such unsafe events would require OS support, which remains as our future work. Therefore, to validate the feasibility of our SpanCoverage idea, we need to measure how long it takes an NT-Path to crash (Crash-Latency) or reach an unsafe event (Unsafe-Latency).

We have conducted Crash-Latency and Unsafe-Latency measurements on all applications used in our experiments (Section 5). In each experiment, we spawn every non-taken path with zero exercise count as an NT-Path and execute it until it either (1) crashes, (2) reaches an unsafe event, (3) reaches the end of the program, or (4) has executed a maximum threshold of instructions (1000 in our experimental setup). In these experiments, NT-Paths are executed without applying any of the variable-fixing techniques described in Section 4.4.

Figure 6 shows the cumulative distribution of Crash-Latency and Unsafe-Latency for three representative applications: SPEC95 benchmark 099.go and two SPEC2000 benchmarks (164.gzip and 175.vpr). The results with the other
applications are similar. In the figure, the cumulative distribution curve of Crash-Latency (or Unsafe-Latency) shows the percentage of NT-Paths that crash (or reach an unsafe event) before executing a given number of instructions.

Our Crash-Latency and Unsafe-Latency statistical results indicate that most NT-Paths can execute for a reasonably long time, allowing the underlying dynamic bug detection tool to detect bugs on these non-taken paths. As shown on Figure 6, in all three applications, 65–99% of the NT-Paths can execute at least 1000 instructions without interrupted by unsafe events or crash. In particular, only 0.5% NT-Paths in go stop before executing 1000 instructions. For gzip and vpr, majority of NT-Paths stop before executing 1000 instructions. Therefore, if we had an OS support to sandbox unsafe events, more than 90% of NT-Paths would execute at least 1000 instructions.

In the example shown in Figure 1, if a non-bug-triggering input is used, the list is not NULL and the execution will bypass line 103. If SpanCoverage selects this non-taken path as a NT-Path for exploration, this NT-Path can execute till the null-pointer dereference bug (line 211) manifests itself, even though the value list is inconsistent with the path taken. Of course, in some other cases, such inconsistency may interfere with bug detection in NT-Paths. Therefore, SpanCoverage also employs inconsistency fix techniques to address this problem (Section 4.4).

4 SpanCoverage Architecture

4.1 Architecture Overview

To provide the desired functionality of increasing the path coverage for dynamic bug detection tools, the SpanCoverage architecture needs to address the following major issues: (1) Which branch, when and how to spawn an NT-Path? (2) How to sandbox the side effects of an NT-Path? (3) How to reduce overheads imposed by executing NT-Paths? (4) How to reduce false positives caused by state inconsistency in an NT-Path execution?

Figure 7: SpanCoverage hardware architecture (The VTag is a single bit volatile tag. With the CMP optimization, the VTag is a 8-bit version tag denoting the corresponding path ID). Additionally, with the CMP optimization, there is also the TLS support to copy registers from the primary core to an idle core and also keep track dependencies among concurrently executed paths.)

As shown on Figure 7, the standard configuration of SpanCoverage requires only simple hardware extensions: (1) extending the branch target buffer (BTB) with 2 four-bit exercise counters, one for each edge, to record the number of times this edge is executed as either a taken path or an NT-Path; (2) simple checkpoint and rollback support for sandboxing NT-Paths’ updates; and (3) a special predicate register to support consistency fixes. For simplicity, we currently assume that the monitored program is a sequential program. We will extend the design to multi-threaded
programs in the future.

There are two methods to sandbox an NT-Path’s memory updates in the standard configuration of SpanCoverage. One is to buffer updates in a store buffer inside the processor, as used in many previous studies for other purposes [31, 1]. Another method is to buffer updates in caches. The tradeoff between the two methods is that the former is simpler and faster, while the latter can buffer more updates. SpanCoverage takes the second approach: buffering updates in caches because it allows NT-Paths to execute for much longer time to expose bugs. More specifically, a single bit VTag is associated with each cache line in L1. If this bit is set, this cache line contains updates by an NT-Path. So when this NT-Path is terminated, all cache lines whose VTags are set to 1 are invalidated.

A special memory area pointed by the Monitor_memory_area register in each processor is not sandboxed. This memory area is used to store error-reports made by the underlying bug detection tool during an NT-Path execution. When an NT-Path ends, all its side effects except those made to this memory area are discarded.

Execution of NT-Paths may cause state inconsistency, which may result in false positives and negatives in bug detection. To minimize this effect, SpanCoverage uses compiler-inserted predicated instructions to fix some key variables before an NT-Path is executed (Section 4.4). The predicate register is used to support such instructions.

With the CMP optimization option, SpanCoverage leverages several hardware extensions that have already provided by the baseline TLS architecture [6, 34], namely cache versioning (a version tag to each L1 cache line), fast register copy from one core to another, lazy commit, cache gang-invalidation, and special logic to keep track of data dependency and commit/squash order. The VTag is an eight-bit version tag. Additionally, SpanCoverage also requires a few modifications to the baseline TLS architecture. Section 4.3 discusses the modifications in more detail.

4.2 The Architecture of SpanCoverage with the Standard Configuration

NT-Path Selection SpanCoverage selects a non-taken path to execute as an NT-Path based on this path’s exercise count. If its exercise count is smaller than a given threshold (NTPathCounterThreshold), it is selected. To perform such selection, SpanCoverage checks the exercise counters stored in the BTB. The threshold can be larger than 1 because, even if a specific branch edge is already executed before, due to different execution history at preceding branches, some new paths that consist of this branch edge with other edges may not be tested yet.

The exercise counters are updated during both taken-path and NT-Path execution. They are periodically reset to zero (per CounterResetInterval binary instructions) to support long-running programs. The rationale for doing this is because after some period of warm-up time, the set of paths executed by the program usually becomes stable and so SpanCoverage may no longer spawn any NT-Paths. However, since the application, system or hardware states still keep changing, some new sequences of branch edges may appear as non-taken paths. Resetting exercise counters to zero allows SpanCoverage to check these possibly new non-taken paths, even though all branch edges in these paths have already been covered.

SpanCoverage works well with the existing implementations (both cache-based and hash-based) of BTBs. In a cache-based BTB implementation, a miss in the BTB can simply be treated as if the corresponding non-taken branch edge has an exercise count of zero so that it is selected as an NT-Path. In a hash-based BTB implementation, if a collision happens, it may result in not spawning a non-taken path as an NT-Path. Both cases do not affect the correctness of program execution. It only affects the coverage slightly.

NT-Path Sandboxing After SpanCoverage decides to explore a non-taken path as an NT-Path, it checkpoints the architectural registers as well as the program counter and redirect the execution to the NT-Path. The register
checkpoint are performed by saving the mapping into a special buffer inside the processor in a way similar to [1]. Before the redirection, the pipeline is flushed. Therefore, both checkpointing, pipeline flushing, and execution redirection can result in some performance overhead.

During an NT-Path’s execution, all memory updates except those to the monitor memory area are sandboxed within the L1 cache. As mentioned in section 4.1, each cache line in L1 is tagged with a Volatile bit. At the first write to a cache line by a NT-Path, if the destination line is dirty, it is first written into L2 to make sure updates previously made by the taken path is committed. After the write operation, the line in L1 is marked as “volatile” so that this line should be invalidated when the NT-Path terminates.

**NT-Path Execution and Termination** A NT-Path’s execution does not stop at the first encountered branch instruction. Instead, it may execute many branch instructions. For simplicity, at each branch instruction in an NT-Path, our SpanCoverage prototype explores only taken edges in a NT-Path execution. This is because exploring non-taken edges in a NT-Path may worsen the inconsistency problem.

Figure 8 compares 164.gzip’s crash latency between our simple policy described above and an alternative policy that chooses the less frequently exercised edge at every branch instruction in an NT-Path. Our experiments show that this scheme can slightly enlarge the branch coverage by 2%, but the crash latency results become much worse. The percentage of NT-Paths crashed before executing 1000 instructions increases from 5% to 16%. Due to this reason, SpanCoverage does not follow non-taken edges in an NT-Path.

When an NT-Path is executed, the L1 cache displacement policy can only select cache lines whose VTag bits are zero. When no more such line is available, the NT-Path is terminated.

Additionally, an NT-Path is also terminated when any of the following conditions hold: (1) the NT-Path has executed long enough (i.e. has executed MaxNTPathLength instructions); (2) the NT-Path crashes; and (3) the NT-Path reaches an unsafe event such as a system call that can no longer be sandboxed. The first condition prevents a NT-Path from occupying too many resources and introducing too much performance overhead. The second condition is obvious since the NT-Path cannot continue its execution. When an NT-Path crashes, it is squashed and the exception that caused the crash is not delivered to the OS. The third condition is necessary because an NT-Path’s side-effects should not be visible to the program’s normal execution. To sandbox unsafe events requires support from OS, which remains as our future work.

When squashing an NT-Path, SpanCoverage rolls back the system status by gang-invalidating all L1 cache lines whose VTags are set. These operations can be done in a handful of cycles using inexpensive custom circuitry[26].

**4.3 CMP Optimization**

**NT-Path Selection, Execution and Termination** As shown on Figure 9(b), with the CMP optimization option, an NT-Path is executed concurrently with the corresponding taken path, and multiple NT-Paths spawned at different branch instructions may be executed simultaneously. After branch α, the NT-Path, D, is selected and executed in an idle core, while the primary core continues executing the taken path, B. The exercise counts for both edges of branch α in the primary core’s BTB are incremented by 1. At a subsequent branch instruction b, the corresponding non-taken path, E, may be spawned as an NT-Path if its exercise count is smaller than the specified threshold.
$NTPathCounterThreshold$. For convenience of description, we call path $A$ as the parent path of $B$ and $D$, and $B$ as $D$'s sibling taken path.

Once an NT-Path is selected, SpanCoverage copies all register values from the primary core to the selected idle core before the NT-Path begins executing there. If no idle processors are available, this NT-Path is temporarily queued using a free thread context.

To avoid spawning too many outstanding NT-Paths, which can incur high resource contention, we use a threshold called $MaxNumNTPaths$ to limit the maximum number of outstanding NT-Paths. A non-taken path is not spawned as an NT-Path when there are already $MaxNumNTPaths$ of outstanding NT-Paths, even though its exercise count is smaller than $NTPathCounterThreshold$.

During an NT-Path's execution, all memory updates except those to the special monitor memory area are sandboxed within the L1 cache. To support it, an eight-bit ID is assigned to each taken path and NT-Path. Each cache line in L1 is tagged with a path ID indicating its version. ID zero is reserved to indicate committed data. Once a path is committed or squashed, its ID can be recycled. When an NT-Path is squashed, all L1 cache lines tagged with its ID are gang-invalidated. When a taken path is committed, all cache lines with its ID are committed by changing the VTags to zero lazily [34]. Updates to the monitor memory area are always tagged with ID zero.

**Modifications to TLS** SpanCoverage modifies the traditional TLS in the following ways:

(1) *Data Dependency*. The traditional TLS is used to speculatively execute sequential programs in parallel, so it follows a strict linear order for data dependency (Figure 9(c)). In contrast, SpanCoverage needs to follow a tree-structured partial order. At each branch, both the taken path and non-taken path of a branch may be executed concurrently, they should read data produced or propagated by its parent path. In other words, any updates made after its parent path should be invisible to them. As shown on Figure 9(b), $B$ and $D$ should read data generated either by or before its parent path $A$. Specifically, a taken path always reads the most recent version in from the primary processor’s L1, or from L2 if no version is available in L1. It should not read data produced by an NT-Path. Since an NT-Path is always spawned from a taken path, it fetches data from either its own L1, the primary core’s L1, or the L2 cache, and in this order. It should not read data produced by another NT-Path.

(2) *Commit/Squash Dependency*. In the traditional TLS, threads are committed following the sequential order of the program. In contrast, due to the above data dependency constraint, SpanCoverage cannot commit a taken path (e.g. path $C$) until its parent taken path (e.g. path $B$) is committed and its sibling NT-Path (e.g. path $E$) is squashed. Otherwise, its sibling NT-Path (e.g. path $E$) may read data generated by this taken path (e.g. path
violating the data dependency described above. In other words, in order to commit, a taken path needs two
tokens, a commit-token from its parent taken path and a squash-token from its sibling NT-Path. Even after these
two conditions satisfied, a taken path does not commit until it has to: when one of its dirty lines is being displaced
from L1 to L2. Since L2 caches do not have any VTags, any writes in L2 are committed. Therefore, when such an
event happens, this taken path needs to commit right away. To avoid stalling this taken path’s commitment, its
sibling NT-Path is squashed immediately.

(3) *Sequential Semantic Checks.* The traditional TLS needs to check for possible write-read and write-write viola-
tions to ensure the sequential semantics, whereas SpanCoverage never needs to do such checks. Since SpanCoverage
never executes any taken path speculatively, all taken paths can always commit. Since the main program is executed
sequentially in the primary processor, there is no data dependence violation among taken paths. In the example
shown on Figure 9(b), path C starts after path B has already finished its execution. Therefore, path C is not
speculative. The squash of an NT-Path is not because of a sequential semantics violation, but because of the need
to discard the side effects of this NT-Path since this path is *not* supposed to happen.

4.4 Fixing State Inconsistency to Reduce False Positives in NT-Paths

As mentioned in Section 3.2, NT-Paths may introduce state inconsistency that may result in false positives in
bug detection. To address this problem, SpanCoverage performs some simple variable fixes. These consistency fixes
can also help to detect more bugs that are related to the values of the condition variables (or other variables on
which the condition variables depend). As shown in our experiments (Section 6.1), fixing consistency not only helps
to prune false positives introduced by SpanCoverage, but also helps to find two more bugs.

To completely fix all state inconsistency in an NT-Path execution is very difficult because it requires fixing not
only the condition variables that set the branch direction but also the variables on which the condition variable
depends. Fortunately, previous works on branch mutation [11, 46](Section 2.2), our feasibility analysis (Section 3.2),
and our experimental results (Section 6) indicate that it is enough to fix some key variables before executing an
NT-Path. Therefore, SpanCoverage only fixes the condition variables of the branch corresponding to a given NT-
Path. We will investigate using program analysis, such as theorem prover, or reverse execution with dynamic slice
information[11], to perform more sophisticated consistency fix in our future work.

4.4.1 How to Fix Key Variables?

To ensure that the branch condition in an NT-Path holds, there can be many ways to fix the condition variables.
For example, in an inequality comparison condition such as $x < 5$, we can make the condition $x < 5$ true in the
NT-Path by changing $x$’s value to anything that is smaller than 5 to force executing the TRUE edge. To make the
fix more accurate, one method is to rely on static analysis and value-invariants inference provided by tools such as
DIDUCE [13]. In the above example, based on either static analysis or value-invariants inference, we may find out
that $x$ is an integer within the range of 4 to 10. Therefore, we can set $x$ to be 4 using a predicated instruction. Of
course, if no information is available, we can randomly set this variable to any value that would satisfy the branch
condition. In our current implementation, if only one value satisfies the condition (e.g. an equality condition), this
variable is fixed to be this value in the NT-Path. If a range of values satisfy the condition (e.g. an inequality
condition), the variable is fixed to be either zero if zero is in the range, or an integer in the range that is closest to
zero otherwise.

Fixing pointer variables is more challenging. For example, the condition of a branch may be based on whether a
pointer p is null or not. Suppose that p is null in the execution. If SpanCoverage decides to execute the wrong path as an NT-Path, the pointer p needs to point to some real data. Otherwise, it may crash or introduce many false positives in bug detection.

To address the above problem, SpanCoverage relies on the compiler to create a blank data structure for each data type, including both basic data types and user-defined structures, at the beginning of the program execution. In the above example, SpanCoverage would fix the problem by setting the pointer p to point to the blank data structure of the correct type. Even though the blank data structure does not have any meaningful content, our experiments show that this simple fix is effective to eliminate many false positives (Section 6.1).

4.4.2 Who Fixes Key Variables?

Since variable fixing is program-dependent, it is not feasible to rely on hardware to fix it. Therefore, SpanCoverage uses a compiler to insert variable-fixing instructions after each branch edge that could possibly spawn as an NT-Path. There are two ways to estimate whether a path can become an NT-Path: static analysis, and profile-based analysis. In our current prototype implementation, we simply assume that the two edges of any branch can be NT-Paths. Therefore, variable-fixing instructions are inserted in both edges.

4.4.3 Where to Execute Variable-Fixing Instructions?

Variable-fixing instructions should be executed only at the beginning of an NT-Path, not in a taken path or in the middle of an NT-Path execution. Clearly, taken paths do not require any variable fixing. For the latter case, since SpanCoverage always follows taken edges in an NT-Path, an edge executed in the middle of an NT-Path is a correctly-taken one instead of a being-forced one. Therefore, it does not require any variable fixing.

To provide the above functionality, variable-fixing instructions are *predicated* in a way similar to previous work [18, 24] so that these instructions are executed only at the entrance of an NT-Path. In the standard configuration, the predicate register is true only at the time when the checkpoint is taken and the program counter is just redirected to the entry of an NT-Path. After rollback, the predicate register is set to be false so that any fetched predicated instruction in the taken path behaves like a NOP. In the CMP optimization, the predicate register is only true for an idle core which is selected to execute an NT-Path. After the first non-predicated instruction in the NT-Path, the predicate register in this idle core is set back to be false.

4.4.4 An Example

To illustrate our idea, Table 1 gives an example how the variable-fixing is performed before executing an NT-Path. Suppose x’s value is 4, and lines 2–3 (function foo) are spawned as a NT-Path. To fix the condition variable x in this NT-Path’s execution, the compiler inserts a predicated instruction (T)x = 2 to make the variable equal to 2. As a result, executing this path as a NT-Path is less inconsistent and thereby the possibility for extra false positives (introduced by SpanCoverage) in bug detection along this NT-Path is reduced. If this path is executed as a taken path, the predicate is false. Therefore, the predicated instruction behaves like a NOP.

5 Experimental Methodology

**Simulator** To evaluate SpanCoverage, we have built an execution-driven simulator that models a 4-core CMP augmented with SpanCoverage functionality. Each core is 4-issue wide and has a private L1 cache. The chip has a shared L2 cache. The parameters of the architecture are shown in Table 2. We use only one core when evaluating the standard configuration of SpanCoverage and use all 4 cores for the CMP optimization option. Since spawning a
### Table 1: Key variable fixing in SpanCoverage. \( (T)x=2 \) is a predicated instruction, and is executed only at the entrance of an NT-Path.

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Original Code</th>
<th>SpanCoverage Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>if ((x==2)) {</td>
<td>( \text{if } (x==2) { )</td>
</tr>
<tr>
<td></td>
<td>( (T)x=2; )</td>
<td>( (T)x=2; )</td>
</tr>
<tr>
<td>2</td>
<td>\text{bool odd = var}%x;</td>
<td>\text{bool odd = var}%x;</td>
</tr>
<tr>
<td>3</td>
<td>\text{foo}(\text{var, x, odd});</td>
<td>\text{foo}(\text{var, x, odd});</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 2: Parameters of the simulated architecture.

<table>
<thead>
<tr>
<th>Core Parameters</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU frequency</td>
<td>2.4GHz</td>
</tr>
<tr>
<td>Fetch, Issue, Retire widths</td>
<td>6, 4, 4</td>
</tr>
<tr>
<td>LD, ST queue entries</td>
<td>64, 48</td>
</tr>
<tr>
<td>ROB, I-window sizes</td>
<td>128, 64</td>
</tr>
<tr>
<td>Int, Mem, FP FUs</td>
<td>3, 2, 2</td>
</tr>
<tr>
<td>BTB</td>
<td>2K, 2 way</td>
</tr>
<tr>
<td>Forced-Squash</td>
<td></td>
</tr>
<tr>
<td>baseline</td>
<td>10 cycles</td>
</tr>
<tr>
<td>CMP opt</td>
<td>Spawn</td>
</tr>
<tr>
<td>L1 cache</td>
<td>16KB, 4-way, 32B/line, 3 cycles latency (2 cycles for non-CMP)</td>
</tr>
<tr>
<td>Memory System Parameters</td>
<td></td>
</tr>
<tr>
<td>L2 cache</td>
<td>1MB, 8-way, 32B/line, 10 cycles latency</td>
</tr>
<tr>
<td>Memory</td>
<td>200 cycles latency</td>
</tr>
</tbody>
</table>

NT-Path in the standard configuration needs to flush the pipeline whereas the CMP option does not (but it needs to copy the register files to the idle core), the former incurs slightly larger overhead for NT-Path spawning.

**Evaluated Dynamic Bug Detection Tools** To show the generality of SpanCoverage, we evaluate SpanCoverage using three different dynamic bug detection methods: assertions, software-only dynamic bug detection (CCured), and hardware-assisted dynamic bug detection (iWatcher).

In the software-only dynamic checker category, we choose CCured [7, 29] (version 1.2.5) as the representative because it is very recently proposed and also publicly available. CCured is a hybrid static and dynamic bug detection tool. It first attempts to enforce a strong type system in C programs via static analysis. Portions of the program that cannot be guaranteed by the CCured type system are instrumented with run-time checks. Since we need to use CCured as a dynamic bug detection tool for evaluating SpanCoverage, we use applications with bugs that can be detected only by CCured’s run-time checks (if the buggy code is executed), but not its static checks.

In the hardware-assisted dynamic checker category, we use iWatcher [50] as the representative. iWatcher associates program-specified monitoring functions with monitored memory objects. When a monitored object is accessed, the associated monitoring function is automatically executed to detect bugs. We disable the TLS option in iWatcher to avoid complication.

In our current prototype, we only spawn non-taken paths in non-library, non-instrumented codes and our NT-Path length threshold only counts non-library, non-instrumented codes. This is because most library codes are well tested, and instrumented codes are inserted by dynamic bug detection tools.

**Evaluated Applications** We have conducted two sets of experiments. The first set uses nine buggy applications to evaluate the functionality of SpanCoverage for software debugging. The second set adds 3 bug-free SPEC2000 benchmarks (gzip, vpr and parser) to evaluate the coverage improvement and overheads of SpanCoverage, the performance benefits of CMP and the effects of parameters and alternative design choices (e.g. do not follow correct branch paths along a NT-Path’s execution).

For our first set of experiments, we select nine buggy programs. Four of them, namely Print\_tokens2, Print\_tokens,
Replace, and Schedule, are originally created by Siemens Research [19], and later modified by Rothermel and Harrold [39]. More details about the Siemens-RH benchmarks can be found in [14]. Four programs, bc-1.06, gzip-1.2.4, man-1.5h1 and ncompress-4.2.4 are from the open-source community. 099.go is from the SPEC95 benchmark. Table 3 gives the details about these applications and their bug characteristics.

The focus of our evaluation is to demonstrate how SpanCoverage can help dynamic bug detection tools to detect bugs in non-taken paths, not to compare these three dynamic bug detection methods. For three Siemens-RH benchmarks with semantic bugs, namely Print	okens2, Print	okens and Replace, we use assertions to detect their bugs instead of using CCured or iWatcher (because semantic bugs cannot be detected by CCured, and are hard to detect with iWatcher without program-specific information). We use iWatcher and CCured to detect bugs in Schedule, bc, gzip, man, ncompress and 099.go. To demonstrate the effectiveness of SpanCoverage in increasing coverage of dynamic bug detection, we use inputs that do not expose the tested bugs in these applications without SpanCoverage. These inputs are much more common than bug-triggering inputs.

Additionally, we have also evaluated the effectiveness of SpanCoverage with multiple test cases, each using a different input. In particular, the Siemens benchmark suite, which has been commonly used for software testing in the software engineering field, provides many test cases for each benchmark. We apply SpanCoverage for each test run and show how much SpanCoverage can enhance the branch coverage for each benchmark when the number of test runs increases. For the SPEC benchmarks, we use inputs provided in SPEC. For bc, we have used a production-rule based test case generation technique to generate a large number of random test input cases, in addition to those provided with the bc package.

In our experiments, the threshold MaxNPathLength is 100 instructions for the four small Siemens-RH benchmarks and 1000 instructions for the other large open-source applications and SPEC benchmarks. If we use 1000 for the Siemens benchmarks, almost all NT-Paths will run till the end of the program. The NPathCounterThreshold is set to 5, and the MaxNumNPaths for CMP-option is 32. All results are obtained using this default setup unless otherwise mentioned in Section 6.3, where we study the effects of these parameters.

We should note that, SpanCoverage is designed to improve path coverages. However, since path coverage percentage is hard and often impossible to measure (the total number of possible paths in a program is usually unlimited due to loops and recursive functions), we use branch coverage as the metric in our evaluation. Branch coverage is subsumed by path coverage, so our results may not show the full strength of SpanCoverage in increasing path coverages.

### Table 3: Applications and bugs evaluated.

<table>
<thead>
<tr>
<th>Application</th>
<th>Lines of Code</th>
<th>Bug Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Print	okens</td>
<td>767</td>
<td>semantic bug</td>
</tr>
<tr>
<td>Print	okens2</td>
<td>727</td>
<td>semantic bug</td>
</tr>
<tr>
<td>Replace</td>
<td>563</td>
<td>semantic bug</td>
</tr>
<tr>
<td>Schedule</td>
<td>411</td>
<td>null pointer dereference</td>
</tr>
<tr>
<td>bc-1.06</td>
<td>17,042</td>
<td>heap object corruption</td>
</tr>
<tr>
<td>gzip-1.2.4</td>
<td>8,163</td>
<td>static object corruption</td>
</tr>
<tr>
<td>man-1.5h1</td>
<td>4,675</td>
<td>static object corruption</td>
</tr>
<tr>
<td>ncompress-4.2.4</td>
<td>1,922</td>
<td>stack object corruption</td>
</tr>
<tr>
<td>099.go</td>
<td>29,623</td>
<td>static object corruption</td>
</tr>
</tbody>
</table>

6 Experimental Results

This section first shows the SpanCoverage’s results in bug detection, path coverage improvement with single and multiple inputs, and false positives before and after consistency fixes. Then we present the overhead results for both
the standard configuration and the CMP-optimization option. Finally, we evaluate the effects of various parameters.

6.1 Functionality: Bug Detection Coverage Enhancement

Bug Detection Results As shown in Table 4, out of the total nine buggy applications, SpanCoverage helps the three bug-detection methods detecting the tested bugs in seven applications, even though non-bug-triggering inputs are used for each application. Without SpanCoverage, all of these bug-detection methods fail to detect these bugs due to their path coverage limitation.

The main reason for the above results is that SpanCoverage extends the detection coverage to include non-taken paths in the monitored run. In particular, SpanCoverage improve the branch coverage of the baseline by 35–100%, as shown in Figure 10. For example, in Print_tokens, the original monitored execution only exercises 48.3% branch edges of the whole programs. With SpanCoverage, this percentage increases to 92.2%. As a result, bugs that occur on the 43.9% additional branch edges that are not taken in the monitored run can be detected by the underlying bug-detection methods. On average, SpanCoverage increases branch coverage from 36.1% to 60.2%.

For two applications (gzip-1.2.4 and ncompress-4.2.4), SpanCoverage cannot help catching the bug with inputs that do not expose the tested bugs. The reason is that the bugs in these applications all occur on taken paths but the particular variable values in the monitored run do not trigger the bug. In other words, these bugs cannot be detected by CCured due to value coverage instead of the path coverage problem. Therefore, SpanCoverage cannot help in these cases. To address this problem would require a solution like the one proposed by Larsen and Austin [21].

<table>
<thead>
<tr>
<th>Dynamic Tools</th>
<th>Application</th>
<th>Bug Detected?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Baseline</td>
</tr>
<tr>
<td>Assertions</td>
<td>Print_tokens</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Print_tokens2</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Replace</td>
<td>No</td>
</tr>
<tr>
<td>CCured</td>
<td>Schedule</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>bc-1.06</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>gzip-1.2.4</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>man-1.5h1</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>ncompress-4.2.4</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>099.go</td>
<td>No</td>
</tr>
<tr>
<td>iWatcher</td>
<td>Schedule</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>bc-1.06</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>man-1.5h1</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>099.go</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 4: The bug-detection results of SpanCoverage with buggy applications using inputs that do not expose the tested bugs. Baseline means no SpanCoverage.

Figure 10: SpanCoverage’s improvement on branch coverage
Coverage Improvement with Multiple Test Cases  To understand more on SpanCoverage’s effects on path coverage improvement, we conduct an evaluation using multiple test cases, with each test case using a different input. As we explained in Section 5, the test cases for the Siemens benchmarks are fairly complete as these benchmarks are commonly used for software testing evaluation. As shown in Figure 11, the branch coverages increase gradually with the number of test cases for all Siemens benchmarks, and reach close to 81.0–89.7% in the baseline case.

SpanCoverage increases the number of covered branch edges by 10.6–126.2% for various numbers of test cases. For example, with the Replace benchmark, SpanCoverage increases the branch coverage from 29.0% to 65.6% for one test case. The improvement becomes less pronounced when the number of test cases increases. With 12 test cases, the baseline achieves 80% branch coverage of the program. Beyond 12, with more test cases, branch coverage is only slightly improved to 84.4%, and the remaining 15% of branch edges cannot be exercised. However, with SpanCoverage, these remaining 15% branch edges can be exercised as NT-Paths so that bugs in these paths can be potentially detected by dynamic bug detection tools. Note that this last 10–15% coverage is always the most difficult in software testing. Because Siemens benchmarks are small, we are able to manually examine these covered non-taken paths and validate that almost all of them are not dead edges, i.e. they are reachable with a particular input and state combination.

Since bc and SPEC benchmarks were not proposed for software engineering studies, their branch coverages do not necessarily increase with the number of test cases. Since the size of these applications are much larger, designing a high-coverage input test suite is very difficult. Only 099.go from SPEC95 reaches 87.9% branch coverage in the baseline case. For gzip, we have tried many different test inputs, but they do not help increasing the branch coverage. In contrast, SpanCoverage provides an easy way to achieve higher coverage automatically without little efforts from programmers. It improves the cumulative branch coverage of bc, gzip and vpr from 28.5–57.2% to 45.1–75.4%.

False Positives  Our results also show that our consistency fixing techniques described in Section 4.4 are very effective in pruning false positives and detecting more bugs. As shown in Table 5, fixing key variables is able to reduce the number of false positives from an average of 10 to only 3, and help detect the bugs in Print.tokens and man (2 out of 7 applications). In particular, it is very effective for go and man. It reduces the number of false
Table 5: Effects of false-positive pruning by fixing key variable values positives from 83 to only 20 for go, and from 7 to 0 for man, respectively. After the consistency fixing, SpanCoverage introduces only 3 false positives on average.

However, even after SpanCoverage variable-fixing, some false positives still remain. For example, go’s remaining 20 false positives are caused by its intensive array operations and complicated variable correlation. These false positives can be further pruned with some extensions such as skipping initialization phases, fixing correlated variables in addition to key variables, etc, which remains as our immediate future work.

### 6.2 SpanCoverage Overhead

As shown in Figure 12, with the standard configuration, SpanCoverage imposes an average of 50.9% overhead for tested applications. In general, the overheads with SPEC benchmarks are relatively small, only 4.5-23.2%, because they spawn fewer NT-Paths than the open-source real applications. Such overhead is significantly smaller than that imposed by dynamic bug detection tools (usually up to 40-100 times slowdowns). Therefore, SpanCoverage’s overhead is acceptable.

With the CMP optimization, SpanCoverage incurs less than 10% overhead for all applications. Such results indicate that, if we can leverage a TLS support in CMPs, SpanCoverage’s overhead can be significantly minimized. The main reason for such a low overhead is that this optimization executes NT-Paths on idle cores concurrently with taken paths.

Table 6 shows the detailed performance results with SpanCoverage. The overhead of the standard SpanCoverage

<table>
<thead>
<tr>
<th>Application</th>
<th>#Spawns per MInst</th>
<th>Avg.NT-Path length</th>
<th>#NT-Path Inst per MInst</th>
<th>#Predicated Inst per MInst</th>
<th>L1 Miss (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>6.0</td>
<td>9764</td>
<td>59K</td>
<td>106K</td>
<td>5.52</td>
</tr>
<tr>
<td>175.vpr</td>
<td>9.3</td>
<td>979</td>
<td>9K</td>
<td>103K</td>
<td>3.64</td>
</tr>
<tr>
<td>195.parser</td>
<td>23.1</td>
<td>986</td>
<td>22K</td>
<td>136K</td>
<td>1.96</td>
</tr>
<tr>
<td>bc-1.06</td>
<td>669.3</td>
<td>556</td>
<td>372K</td>
<td>45K</td>
<td>0.39</td>
</tr>
<tr>
<td>man-1.5h1</td>
<td>821.9</td>
<td>537</td>
<td>688K</td>
<td>41K</td>
<td>1.17</td>
</tr>
<tr>
<td>099.go</td>
<td>364.3</td>
<td>1029</td>
<td>375K</td>
<td>75K</td>
<td>2.32</td>
</tr>
</tbody>
</table>

Table 6: SpanCoverage detailed performance results. MInst denotes one million instructions counting both NT-Path and taken-path instructions. By default, SpanCoverage means our standard configuration unless specified by CMP-Opt.
comes from four main sources: (1) NT-Path spawning and termination overheads, including register checkpoint and restore, cache line invalidation, pipeline flushing. (2) NT-Path execution. The first one is reflected by the NT-Path spawning frequency shown in Table 6, i.e., the number of spawns for every million instructions. The second is shown by the average NT-Path length and NT-Path instruction proportion to the total number of instructions, in the Table 6 column 3 and 4. In general, in the standard configuration, the higher the spawn frequency and the longer the NT-Paths, the higher the percentage overhead imposed by SpanCoverage. For example, go, bc and man have higher spawning frequencies than the SPEC benchmarks. Therefore, SpanCoverage imposed much more overhead on these three applications. (3) Execution of predicated instructions. Although those instructions become NOPs in the taken path, many of such instructions still incur visible overhead. The frequency of predicated instructions executed are shown in Table 6. (4) Slightly higher L1 miss rate due to cache pollution by NT-Paths also contribute to SpanCoverage's overhead.

For some applications such as gzip, bc, man and go, CMP-optimization successfully decreases the overhead from 23.2%–142.4% to less than 9.9%. However, for parser and vpr, the optimization effect is less pronounced and even negative. The reason is, while CMP can increase the level of parallelism, it also imposes some extra overheads, such as increased L1 cache latency due to cache versioning, and register copying from one core to another. Not all of these overheads can be hidden by parallelism. As a result, if the original sequential overhead is small, the parallelism benefit are outweighed by the extra overheads to support the parallelism.

6.3 Effects of Parameters

In this section, we evaluate the effects of the two parameters, namely MaxNTPathLength and MaxNumNTPaths on the standard and CMP-optimized SpanCoverage overheads.

As shown in Figure 13(a)(c), with the standard configuration, SpanCoverage's overhead increases gradually with MaxNTPathLength and NTPathCounterThreshold. This is quite expectable because when MaxNTPathLength increases, many NT-Paths that do not crash or reach an unsafe event early would run longer; and when NTPathCounterThreshold increases, more non-taken paths are selected for exploration as NT-Paths.

However, with the CMP optimization, as shown in Figure 13(b)(d), the overhead increases only slightly with these parameters. Changing NTPathCounterThreshold from 1 to 20 only introduces 1.4–2.3% more overheads on all three applications. The effect of MaxNTPathLength on performance is even smaller. The reason is, the CMP optimization runs NT-Paths in parallel with taken paths, thus executing NT-Paths longer will not significantly effect the performance on the primary processor. The NTPathCounterThreshold will increase the overhead a little due to higher spawning and squashing overhead.

The insensitiveness of CMP optimization to these parameters provides SpanCoverage good potential to carry out more intensive NT-Path exploration and more path coverage extension, hence more help to dynamic bug detection.
7 Related Work

Our work builds upon many previous studies. Due to space limitation, we briefly describe closely related work that is not discussed in early sections.

(1) Multipath Branch Execution Several previous studies [17, 20, 45, 1, 2] have used the technique of running both paths of a branch to reduce the branch mis-prediction penalty, and some of them also explore spare contexts in a SMT to reduce overhead. We execute non-taken paths on idle processors to increase the coverage of software bug detection. Therefore, we need to address different issues: (1) Our NT-Paths need to run much longer because our purpose is to detect bugs in NT-Paths. In contrast, the multipath architectures need to run multiple branch edges only until the branch is resolved. Therefore, SpanCoverage needs to buffer many more side effects than the above works. (2) SpanCoverage needs to keep track of exercise count of every branch edges in order to select NT-Paths to avoid incurring large overhead. (3) Our NT-Paths are spawned after a branch is resolved, whereas multipaths are executed before a branch is resolved. (4) Our NT-Paths are always squashed, whereas in the multipath architectures only the mis-predicted path is squashed. (5) SpanCoverage needs to address the state inconsistency problem because it may introduce false positives and false negatives in bug detection.

(2) Adaptive Statistical Profiling Although SpanCoverage bases the NT-Path selection on a path’s exercise counts, it is fundamentally different from a prior work [16] by Hauswirth and Chilimbi, which tries to reduce dynamic monitoring overhead via statistical sampling but still ensures rarely exercised paths are sampled at least once. It tries to take a sample in a taken path that has low exercise count. In contrast, SpanCoverage tries to execute a non-taken (a path which is NOT supposed to take according to the branch condition). In other words, while both SpanCoverage and their work [16] consider the exercise count of a branch edge, SpanCoverage forces execution of a non-taken path but their work still follows the program’s normal execution. Therefore, SpanCoverage can address the path coverage limitation for dynamic bug detection tools, whereas theirs cannot. In addition, as acknowledged by the authors themselves, their technique works only for detecting memory leaks, but not for other types of bugs such as buffer overflow, memory corruption, etc [16]. In contrast, SpanCoverage can work with any dynamic bug detection tools to detect any types of bugs.

(3) Branch Mutation Just like SpanCoverage forces some branches to follow a wrong path, some other works also use branch mutation for other purposes. In fault injection area, studies [3, 37] use branch mutation to evaluate the impact of faults in systems. In software engineering, delta debugging [48] changes variable values on-the-fly to force the control flow down different paths in order to find a correct execution that are most similar to a known wrong one to do delta debugging (finding the differences between the two) in postmortem bug analysis. Our work uses branch mutation with architectural support for a very different goal: increasing the coverage for dynamic bug detection tools to detect bugs in non-taken paths and thereby have different challenges and design issues.

(4) Dynamic Bug Detection Many tools have been proposed for dynamic execution monitoring. Well-known examples include Purify [15], Valgrind [40], CCured [7] and others [4, 33, 23]. SpanCoverage would benefit almost all such tools by increasing their path coverage in each monitored run.

(5) Architectural Support for Debugging Recently, many researchers in the architecture community have shown the effectiveness of providing architectural support for software debugging. Just to name a few, ReEnact [35], Flight Data Recorder [47], iWatcher [50], AccMon [49], and others [44] are examples of architectural innovations to
improve software robustness. While these studies have demonstrated that architecture support can more accurately and efficiently detect certain types of bugs or provide more information for postmortem analysis, none of them addresses the path coverage problem of commonly-used dynamic bug detection methods. SpanCoverage can benefit them in detecting bugs in non-taken paths during each single monitored run.

(7) Software Testing Some tools such as PureCoverage [36] can report the statement coverage during monitored executions. Besides the program-based coverage measures such as branch coverage and path coverage, there are also specification-based coverage, fault-based coverage measurement [8], etc. In our work, we focus on increasing the path coverage for dynamic bug detection. Only a few studies [38, 9] have been conducted on how to systematically generate test suites to increase testing coverages. Most work on this direction is based on specification rather than program code, and therefore it is hard to ensure that all program paths are exercised in this way.

(8) Model Checking and Static Analysis Model checking [27] is also related to our work because it explores multiple paths to verify certain properties such as deadlock or data-race free. But model checking is usually done statically and is mostly based on specification or program annotation, whereas SpanCoverage is a dynamic approach and requires no specification or annotation. Recently, “exploring” both branch edges simultaneously is also studied in static program analysis to decrease analysis complexity for codes written in special programming language [12]. Different from these works, SpanCoverage is dynamic with totally different goals.

8 Conclusions and Future Work

This paper addresses the fundamental problem of path coverage for the commonly-used dynamic bug detection method. Specifically, we have proposed an innovative, automatic, general and low-overhead approach, called SpanCoverage, which uses a simple architectural support to dynamically increase the path coverage of dynamic bug detection and thus reduce the amount of efforts required for software engineers to design and implement test cases to cover these extra paths in testing. We have evaluated SpanCoverage in an execution-driven simulator using three different dynamic bug detection methods: (1) assertions, (2) software-only dynamic checkers (CCured) and (3) hardware-assisted dynamic checkers (iWatcher). Our experiments with nine buggy programs using inputs that do not expose the tested bugs show that SpanCoverage is able to help these tools detect seven out of nine tested bugs that would otherwise not be detected. This is because SpanCoverage increases the branch coverage of the monitored execution from 36% to 60% on average. Our results also show that SpanCoverage imposes modest overhead (less than 9.9%) and incurs few (3 on average) false positives with only simple consistency fix.

Even though our work makes a significant first step (to the best of our knowledge) in addressing the fundamental path coverage problem in dynamic bug detection, our study has several limitations that we plan to address in the future. First, SpanCoverage cannot help dynamic bug detection tools to detect bugs that (1) are due to value coverage; (2) require spawning an NT-Path from an NT-Path; or (3) require deeper state consistency fixes on NT-Paths. We plan to extend SpanCoverage to handle the above cases. Second, due to our simulation infrastructure (it cannot run an OS), we cannot evaluate SpanCoverage with larger server programs. Finally, the current architecture design of SpanCoverage assumes sequential programs and also executes them sequentially. We are investigating extensions to work with multi-threaded programs and speculative execution of sequential programs. It is also conceivable to use software-only solutions such as compilers to provide similar functionality. But it would require a very complicated compiler support to force executing a non-taken path and sandbox all of its side-effects. We plan to investigate this approach and evaluate the tradeoffs between the two.