PathExpander: 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, Urbana, IL 61801
{shanlu,pinzhou,liuwei,yyzhou,torrellas}@cs.uiuc.edu

Abstract

Dynamic software bug detection tools are commonly used because they leverage run-time information. However, they suffer from a fundamental limitation, 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 hardware extension. First, we propose *PathExpander* , a novel design that dynamically increases the code path coverage of dynamic bug detection tools with no programmer involvement. As a program executes, PathExpander selectively executes non-taken paths in a sandbox without side effects. This enables dynamic bug detection tools to find bugs that are present in these non-taken paths and would otherwise not be detected. Second, we propose a simple hardware extension to control the huge overhead in its pure software implementation to a moderate level. To further minimize overhead, PathExpander provides an optimization option to execute non-taken paths on idle cores in chip multi-processor architectures that support speculative execution.

To evaluate PathExpander, we use three dynamic bug detection methods: dynamic software-only checker (CCured), dynamic hardware-assisted checker (iWatcher) and assertions; and conduct side-by-side comparison with PathExpander's counterpart software implementation. Our experiments with seven buggy programs using general inputs that do not expose the tested bugs show that PathExpander is able to help these tools detect 21 (out of 38) tested bugs that are otherwise missed. This is because PathExpander increases the code coverage of each test case from 40% to 65% on average, based on the branch coverage metric. When applications are tested with multiple inputs, the cumulative coverage also significantly improves by 19%. We also show that PathExpander introduces modest false positives (4 on average) and overhead (less than 9.9%). The 3–4 orders of magnitude lower overhead compared with pure-software implementation further justifies the hardware design in PathExpander.

1. Introduction

1.1. The Path Coverage Limitation of Dynamic Bug Detection

Software bug is one of the major causes of system down time and security attacks. 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, 12, 14, 27, 41] 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 [14], DIDUCE [12] and CCured [27], and hardware-assisted dynamic checkers such as ReEnact [31] and iWatcher [41].

Unfortunately, as often pointed out in previous work [7, 18], almost all dynamic bug detection tools suffer from a major limitation: the *path coverage problem*, i.e. they can detect only those bugs which appear in the executed paths. In other words, they require the bug to manifest itself during the monitored runs (runs that are monitored by the dynamic bug detection tool). If a bug is present in a **non-taken** path (a control flow path which is not executed), dynamic tools cannot detect it.

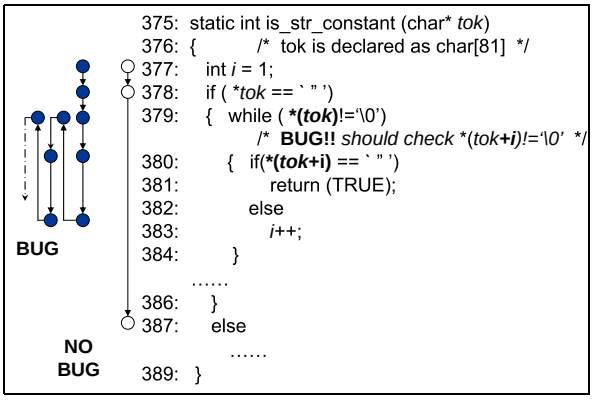

Figure 1. A bug example from Siemens benchmark: bug can be detected only if the solid-circle path is executed.

Figure 1 gives a simple example to illustrate the path coverage problem. This code segment comes from the $print\_tokens2$ benchmark in the Siemens Benchmark Suite [13], a commonly used benchmark suite for software testing and bug detection. This code segment contains a severe bug in Line 379 that can cause a buffer overrun if the solid-circle path (referred as the buggy path) is executed. This bug can be detected by a dynamic buffer overrun detection tools such as Purify [14] if the bug is exposed in the monitored

*This work was supported in part by NSF CNS-0347854 (career award), NSF CCR-0325603 grant, DOE DE-FG02-05ER25688, and Intel gift grant

run. But, unfortunately, to execute this buggy path requires that the string tok starts with a quotation mark and does not contain another quotation mark. Therefore, it needs a special program input that can lead to such special tok string value. For all other inputs, the buggy path is not executed and thus the bug does not manifest. As a result, the underlying dynamic bug detection tool cannot detect this bug. Here, we use this simple benchmark only for illustrative purpose. Real-world examples are often much more complex. For example, the string tok may be an intermediate variable and have complex correlation with the end-user input. In that case, it would be very difficult to generate a bug-triggering input and thus it is very likely this bug will escape from the dynamic bug detection tool.

1.2. Why Path Coverage is So Challenging?

One might wonder why the coverage limitation cannot be solved by simply running as many tests as possible to ensure every path is checked by the dynamic bug detection tool. In fact, the path coverage issue has been an open problem in software testing for decades. There are many challenges:

(1) Incompleteness of test cases: Generating test cases to cover all feasible paths is usually impractical, especially for real-world applications. This is because: (i) the total number of feasible paths in a program is very large: at least exponential to the code size and even *infinite* due to loops and recursive functions; (ii) even for any *one* specific path, generating an input to cover it is *theoretically uncomputable* [33]. Although recently dynamic test generation [10] and symbolic unit test generation [5, 8, 34] have made great progress, they are still limited by the above unbounded computation complexity problem and have many hard-to-handle cases, including loops, dynamic data structures, nonlinear correlations, etc. For example, it is difficult for them to generate an input to trigger the bug in the simple example shown in Figure 1, because that bug triggering path contains a loop.

(2) Dependence on other states: The above problem becomes even harder if the control path also depends on system states (e.g. the time of day), hardware states (e.g. disk status), resource states (e.g. how much virtual memory has been allocated), and application states (e.g. the total number of outstanding requests). While fault injection-based testing can address this problem to some extent, it is usually very expensive and cannot test all possible state combinations.

(3) Human Effort and Testing Efficiency: For real-world programs, each test case can take a large amount of human effort to enforce and a long time to execute. Therefore, the number of cases that can be tested in reality is limited; it is critically important to reduce the number of test cases without sacrificing significantly in overall test coverage [19].

(4) Dynamic Monitoring Efficiency: Dynamic bug detection tools are usually only applied to a small number of selected test runs, because many dynamic tools, such as Purify and Valgrind [28], incur up to 40-100 times overhead [41], too time consuming to monitor hundreds of or more test runs, especially for large software. For example, if each test run without such tools takes 2 hours, using these dynamic checkers 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 *without significantly increasing the execution time* to allow the underlying dynamic bug detection tool to detect more bugs.

1.3. New Opportunity: Hardware Support

Recent rapid advances in computer hardware have led to dramatic performance improvement. Multicore processor is becoming a mainstream technology. This trend provides a unique opportunity to enhance other functionalities in addition to performance, such as software debugging. Recently, several hardware extensions have been proposed to support speculative execution and rollback [2, 11, 24, 30, 31, 32], which also provide opportunities to many functionalities such as increasing path coverage of bug detection proposed in this paper.

1.4. Our Contributions

In this paper, we make one of the first attempts (to the best of our knowledge) to address the fundamental path coverage problem of dynamic bug detection by using a simple extension to existing and emerging hardware. Specifically, we propose an innovative, *automatic*, *low-overhead*, and *general* hardware-assisted framework, called *PathExpander* . PathExpander allows dynamic bug detection tools to detect bugs, that would not be detected otherwise, on non-taken paths (referred as *NT-Paths*). Specifically, PathExpander combines two innovative ideas:

- *Exploring both taken and non-taken paths in a* **single** *monitored run*. PathExpander addresses the path coverage problem in dynamic bug detection by *transparently* and *automatically* executing along *both* the taken edge and the non-taken edge on selected branches at run time. Therefore not only the taken path but also many non-taken paths can be monitored by dynamic bug detectors in each run. The feasibility of our idea is validated through our crash latency measure and other analysis (Section 3.2). The state inconsistency problem in NT-Path execution is addressed by leveraging predicated instructions (Section 4.4).
- *Leveraging hardware support*. While we also implement the above idea in pure software, the huge overhead motivates us to explore hardware extensions to support the PathExpander idea, i.e. to select and execute NT-Paths. *Hardware support not only provides an efficient way to sandbox side effects made by NT-Paths, but also conveniently exploits idle cores in a multicore architecture with speculative execution support to execute NT-Paths with little overhead.*

The main characteristics that distinguish PathExpander are:

(1) Generality. PathExpander makes no assumption about bug types or dynamic bug detection methods (tools). Therefore, it can increase the path coverage of *almost all* dynamic bug detection tools with little modification. Additionally, PathExpander can potentially work with programs written in many different programming languages. In our evaluation, PathExpander successfully helps three different dynamic bug detection methods: (1) a software-only checker, CCured [27]; (2) a hardware-assisted checker, iWatcher [41]; and (3) assertions, to extend their bug detection coverage.

(2) Help Detecting Bugs with non Bug-triggering Inputs. Many bugs require special inputs to manifest during execution. However, with the help of PathExpander, it is possible to expose these bugs even with non bug-triggering inputs, which are usually much more common than those bug-triggering ones. Our experiments with seven buggy programs *using general inputs that do not expose the tested bugs* show that PathExpander is able to help dynamic bug detection tools detect 21 (out of the 38) tested bugs that are missed otherwise (without PathExpander).

(3) Improving Code Coverage. PathExpander significantly increases the code coverage of the monitored execution from 40% to 65% on average. Even when multiple inputs are used for each application, the cumulative branch coverage improvement by PathExpander is still significant, by 19% on average. (Branch coverage is used as coverage metric in our experiments, because path coverage is hard to directly measure.)

(4) Reducing Human Efforts. Since PathExpander significantly increases the path coverage for dynamic bug detection *w*ithout any effort from programmers, it can reduce the large amount of human effort in designing and enforcing various input cases to check bugs on different paths, especially those bugs that require very special inputs to trigger.

(5) Low Overhead. With simple hardware support, PathExpander explores large number (hundreds to thousands in our experiments) of new paths in each run with small overhead (less than 9.9% with the CMP optimization; 3–4 orders of magnitude better than the counterpart software-only implementation). This can greatly increase the efficiency of software testing and debugging, which contribute to 50-70% of software development cost.

(6) Modest False-Alarms. Because PathExpander uses predicated instructions to fix key variables' values before the NT-Path execution, it significantly reduces the number of false positives in bug detection to only a few (4 on average) for our seven tested buggy applications.

(7) Simple Integration with Dynamic Checkers. With hardware support, PathExpander is almost transparent to dynamic checkers and thereby can easily integrate with them (See Section 6.2).

2. Background of 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 [33]. It is easy to measure but is insensitive to control structures, to which many bugs are related. An alternative, **branch coverage** [33], is better in this, but still limited in that it only looks at one branch at a time.

One of the most accurate code coverage metrics is **path coverage** [33]. It focuses on a *sequence* of branch decisions, instead of single branch decision. Path coverage is stronger than statement and branch coverage, because even if a certain branch decision or statement has already been touched, the combinations with other branch decisions (or statements) may have not been 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 coverage are the dominant criteria used in software testing. For the same reason, in our evaluation, we can only show PathExpander's improvement on branch coverage, *even though many of our design decisions are targeted for improving path coverage.*

High testing coverage is always hard to achieve. Especially, as coverage increases, further coverage improvement becomes increasingly difficult. In large software, after around 80–90% branch coverage, each percent of coverage increase requires a huge amount of effort [33]. 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. PathExpander can push bug detection into the remaining 10–30% uncovered program parts *without large overhead and programmer efforts* and thus enhance the software quality.

3. PathExpander Idea

Terminology Definition: In this paper, we use a *branch edge* or *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*, 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. Note that our *design* is oriented by the more accurate path coverage. However, as explained in section 2, path coverage is very difficult to calculate, therefore, in this paper we use *branch coverage* as the code coverage metric in evaluation.

3.1. Idea Overview

The main purpose of PathExpander 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 single monitored run without any extra effort from programmers to design, generate and monitor various special test cases. PathExpander does this by "silently" (without side effects) executing non-taken paths in a hardware or software sandbox.

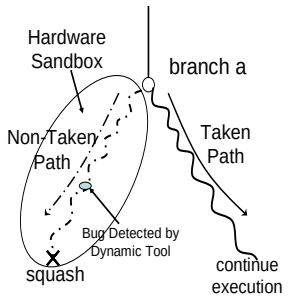

Figure 2. NT-Path in PathExpander (The winding curves denote that both the taken path and the NT-Path may also execute other branch instructions.)

Figure 2 shows the main idea of PathExpander. At a branch a, suppose the actual program execution will follow the right path. PathExpander executes the left non-taken path (referred as an NT-Path) and the right taken path one after another sequentially (in the standard configuration) or in parallel (with the CMP optimization option), as shown in Figure 4. At following branches, the NT-Path

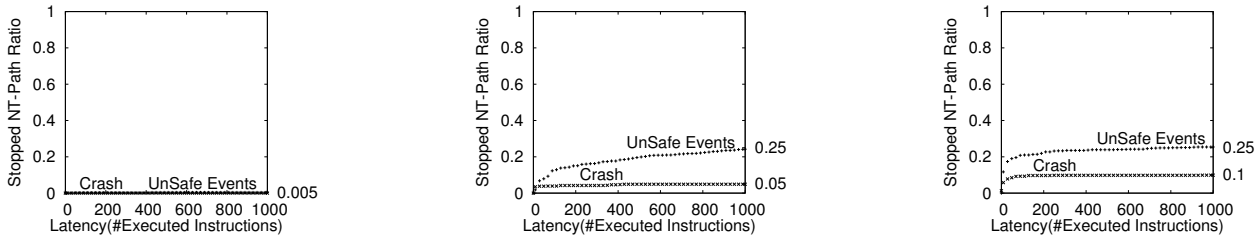

Figure 3. Crash-Latency and Unsafe-Latency statistics (Above cumulative distribution curves show the percentage of NT-Paths that crash or reach an unsafe event *before* executing a given number of instructions.)

follows branch edges based on the actual branch condition while the taken path may spawn new NT-Paths. The NT-Path will be terminated before executing a threshold number of instructions due to the resource competition, state consistency and some other concerns. During the whole life time, all of the NT-Path's memory updates are sandboxed so that they do not influence the execution of the taken path. Other side effects such as I/Os made by an NT-Path are avoided by stopping the NT-Path execution upon such events.

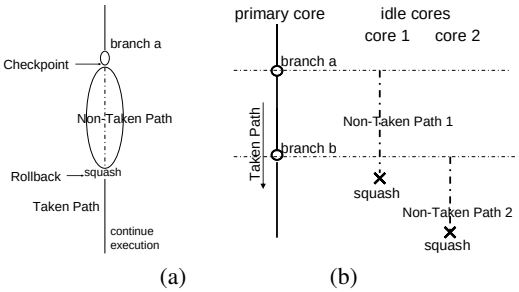

Figure 4. (a) PathExpander standard configuration; (b) PathExpander CMP optimization.

PathExpander provides two options: a standard configuration and a CMP optimization (only available in the hardware implementation). The standard configuration (Figure 4(a)) uses a simple checkpoint-and-rollback scheme. At branch a, a checkpoint is taken and the program execution follows the NT-Path. When the NT-Path terminates, PathExpander rolls back the memory and processor state to the previous checkpoint and resumes the normal execution, i.e. following the taken path. Besides the standard configuration, PathExpander also provides a CMP optimization option (Figure 4(b)) to minimize the overhead. This optimization leverages the CMP architecture to hide the overhead of NT-Path execution. It does so by executing an NT-Path on an idle core while continuing the taken path execution on the primary core. At a subsequent branch on the taken path, a new NT-Path can be explored in another idle core before the previous NT-Path terminates.

PathExpander handles the state inconsistency of NT-Path by simple variable fixing at the entrance of an NT-Path. Take Figure 2 as an example. PathExpander modifies the condition variables used in branch a, so that the outcome of branch a will be flipped to the left edge instead of the original right edge. More details can be found in Section 4.4.

PathExpander does not spawn NT-Paths on all branches, because blind spawning has huge overhead and is unnecessary. Instead, PathExpander selectively executes those non-taken paths whose instructions and branch edges have not been well exercised so far in the monitored execution, because these paths are more likely to have undetected bugs. In our current prototype, we make the selection based on a branch edge exercise counter. In the future, we may extend it to take into account other information, such as memory state and some random factors.

Implementation Choices: Hardware or Software? Like all designs that propose hardware extensions, the trade-off is usually between efficiency and hardware complexity. To evaluate this tradeoff, we have implemented PathExpander using simple hardware extensions (see Section 4) and with pure software (see Section 5). Their tradeoffs are discussed in Section 7.5.

3.2. Feasibility Analysis

State inconsistency problem, i.e. the branch condition variables indicating one branch target but the NT-Path executing the other one, threats the feasibility of PathExpander in two aspects. First, it may lead to an immediate termination of the NT-Path, and therefore prevents NT-Paths from running long enough to reach the potential buggy code region. Second, inconsistency may make the dynamic bug detection tool unable to discover 'real' bugs. We investigate the two issues through our feasibility analysis below.

(1) Termination Latency Analysis An NT-Path may be terminated by a crash (e.g. divide-by-zero, access violations) due to state inconsistency. Fortunately, several recent fault tolerance studies, including the Y-branch study [37] and the Iyer et al's Linux kernel study [9], have examined the state inconsistency effects of injected random *branch mutation* (forcing execution of a non-taken path). Their results show that, in a large percent (30-40%) of the cases, the state inconsistency does not result in any crashes or even silent errors in the following execution, because many of these branches are just optimization paths or short-cuts. These studies provide a good foundation for our work. Actually, their condition is stronger than what is required by PathExpander. Different from them, PathExpander does not execute NT-Paths to the end of the program, and not require NT-Path have the same execution result as taken-path.

An NT-Path may also be terminated by another reason: some special side effects of an NT-Path such as I/O events cannot be sandboxed by PathExpander via either hardware or software. Therefore the NT-Path needs to be squashed when such event occurs. To sandbox such unsafe events requires OS support, which remains as our future work.

To validate the feasibility of our PathExpander idea, we have conducted Crash-Latency (how long it takes an NT-Path to crash)

and Unsafe-Latency (how long it takes an NT-Path to reach an unsafe event) measurements on all applications used in our experiments (Section 6). In each experiment, we spawn an NT-Path at every non-taken branch edge with zero exercise count 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 variable-fixing techniques described in Section 4.4.

Figure 3 shows the cumulative distribution of Crash-Latency and Unsafe-Latency for three representative applications: one SPEC95 benchmark (099.go) and two SPEC2000 benchmarks (164.gzip and 175.vpr). As we can see, 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. The results with other applications are similar.

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. Furthermore, for many applications, such as $gzip$ and vpr, the majority of NT-Paths stop early due to unsafe events. Therefore, if we had an OS support to sandbox unsafe events, more than 90% of NT-Paths may potentially execute up to 1000 instructions.

(2) Can 'real' bugs be detected in an inconsistent state? The answer is 'yes'. First, we should note that the degree of the inconsistency on the NT-Path is small. Such degree is hard to quantitatively measure. However, we can take above crash latency as an indirect measure. Intuitively the more inconsistent the variables are, the more likely the program will crash, therefore our crash latency results indicate that usually the NT-Path inconsistency is not severe.

Second, *PathExpander is designed for bug detection, which has better inconsistency tolerability than software testing*. Usually in testing, if the variables are not completely consistent, the execution result can hardly be used. Different from that, in dynamic bug detection, we do not care NT-Paths' final results since they are sandboxed and discarded anyway. Instead, the goal is to find bugs. Since most bugs are correlated to only a subset of variables, inconsistency in unrelated variables has little interference in the exposure of these bugs.

In the example shown in Figure 1, if the string tok does not start with a quotation mark, the execution will directly jump to line 387 after the check at line 378. If PathExpander selects the non-taken edge to start an NT-Path exploration, this NT-Path will follow through and execute the $while$ loop. Even though there is inconsistency between the tok's starting character and the NT-Path, the bug can still be exposed. In this case, the conditional variable inconsistency will not prevent the bug from manifesting, as long as the $while$ loop is executed.

Undoubtedly, in some cases, such inconsistency may interfere with bug detection in NT-Paths. Therefore, PathExpander also employs inconsistency fix techniques to further address this problem (see Section 4.4).

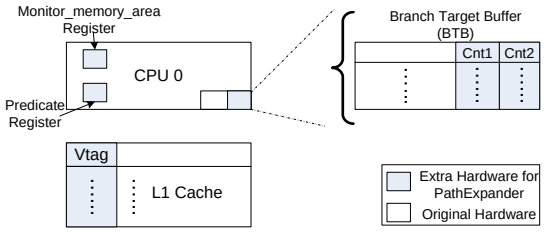

Figure 5. PathExpander hardware architecture (The Vtag in the standard configuration is a single bit volatile tag. With the CMP optimization, it is an 8-bit version tag, denoting the corresponding path ID.)

4. Hardware PathExpander Implementation

4.1. Architecture Overview

To increase the path coverage of dynamic bug detection, PathExpander needs to address the following major issues: (1) What 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 NT-Path execution?

As shown in Figure 5, the standard configuration of PathExpander 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; (2) a special predicate register to support consistency fixes (Section 4.4); and (3) simple checkpoint and rollback support for sandboxing NT-Paths' effects. Specifically, PathExpander buffers NT-Paths' memory updates in L1 cache and uses a 1-bit *Volatile* tag (*Vtag*) associated with each L1 cache line to differentiate cache lines written by NT-Paths from those by taken path.

A special memory area pointed by the *Monitor_memory_area* register in each core 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.

PathExpander's CMP optimization option needs extra hardware support to execute NT-Paths and taken-path in parallel on multicore architectures. Specifically, such hardware support includes fast register copy from one core to another for NT-Path spawn, cache versioning and data dependency tracking for correct isolation and data flow among different execution paths, and special NT-Paths squash mechanism. Most of these supports can be easily got from some emerging advanced architectures. The current PathExpander prototype builds the CMP optimization based on the thread level speculation architecture [6, 30]. The details are discussed in section 4.3.

4.2. PathExpander Standard Configuration

(1) NT-Path Selection PathExpander selects a non-taken branch edge to begin an NT-Path exploration when this edge's exercise count, stored in the BTB, is smaller than a given threshold (*NTPathCounterThreshold*). The threshold can be larger than 1, because even if a branch edge is already exercised, different execution context may still bring up non-tested paths consisting of it and other edges.

The exercise counters are updated during taken-path execution at the entry of an NT-Path. They are periodically reset to zero (per

CounterResetInterval binary instructions) to support long-running programs. The rationale is that during the long running period of such programs, new application, system and hardware states may emerge even after all these branch edges are thoroughly exercised.

The exercise counter of each branch edge is put in BTB (branch target buffer), and read at every BTB access. A BTB miss is simply treated as if the exercise counter is zero.

(2) NT-Path Sandboxing After PathExpander decides to explore a non-taken path as an NT-Path, it checkpoints the architectural registers as well as the program counter in a way similar to previous work [1] and redirects the execution to the NT-Path. During an NT-Path's execution, all memory writes are sandboxed within the L1 cache by leveraging the previously proposed versioned hardware cache [24]. These cache updates are bookmarked by setting the associated 1-bit Vtag. When NT-Path terminates, all cache lines with this bit set will be invalidated.

PathExpander chooses cache instead of store buffer inside the processor [1, 29] to sandbox the NT-Path, mainly because cache can buffer more updates, allowing NT-Paths to execute for longer time to expose bugs.

(3) NT-Path Execution and Termination NT-Path execution does not stop at the first encountered branch instruction. Instead, it may execute many branch instructions. At each encountered branch, NT-Path only explores the taken edge, because otherwise exploring non-taken edges from an NT-Path may worsen the inconsistency problem. This design choice is evaluated by a simple experiment we conducted on 164.gzip. Experiment shows that exploring non-taken edges from NT-Paths slightly enlarges the branch coverage by 2%, but significantly increases the NT-Paths crash ratio before executing 1000 instructions from 5% to 16%, indicating much worse state consistency. Due to this, PathExpander does **not** follow non-taken edges in NT-Paths.

An NT-Path is terminated when any of the following conditions hold: (1) the NT-Path has executed long enough (i.e. reaches *MaxNTPathLength* instructions); (2) the NT-Path crashes; and (3) the NT-Path reaches an unsafe event such as a system call that can not be sandboxed. The first condition prevents an NT-Path from occupying too many resources. The second condition is obvious. 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. When squashing an NT-Path, PathExpander rolls back the system states 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[24].

4.3. CMP Optimization

Different from the standard configuration, the CMP optimization can execute the original taken path and NT-Paths spawned at different branch instructions simultaneously. For example, in Figure 6(a) and (c), after branch a, the NT-Path D is executed on idle core 1, while the primary core continues executing the taken path. At a subsequent branch b, the corresponding non-taken path E may be spawned as an NT-Path and executed on idle core 2 in parallel with the taken path and NT-Path D. For convenience of description, we call code segment A as the *parent* of code segment B and NT-Path D, and B as D's *sibling*.

To spawn an NT-Path in the CMP optimization option, PathExpander copies all register context from the primary core to the selected idle core before the NT-Path begins there. If no idle core is 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 when there are *MaxNumNTPaths* of outstanding NT-Paths.

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 segment and NT-Path. Each cache line in L1 is tagged with a path ID indicating its owner. ID zero is reserved to indicate committed data. Once a taken path code segment or NT-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 code segment is committed, all cache lines tagged with its ID are committed by changing the ID to zero lazily [30]. Updates to the monitor memory area are always tagged with ID zero.

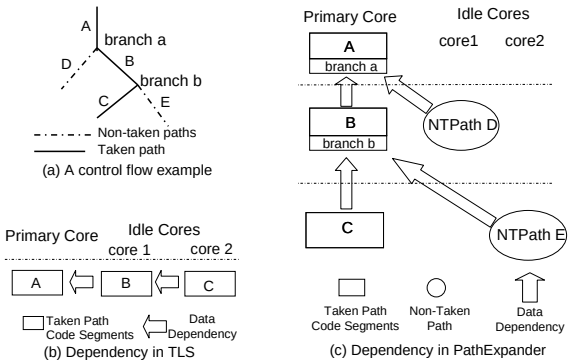

Figure 6. PathExpander dependencies

Since multiple NT-Paths could be executed concurrently with the taken path in the CMP option, special care needs to be provided to ensure that each path reads from the correct version of the data and their memory modifications do not interfere with each other. Such functionality is achieved by maintaining and enforcing data dependencies and commit and squash dependencies among them in ways similar to the thread level speculation (TLS) architecture.

First, the data dependency follows a tree-structured partial order (Figure 6(c)), slightly different from the original linear dependency in TLS (Figure 6(b)). At each branch, both the taken edge (and following code) and non-taken edge (and following code) of a branch may be executed concurrently, and they should read data produced or propagated by their parent code segment. Any updates made after their parent code segment should be invisible to them.

Due to the above data dependency constraint, PathExpander cannot commit a taken path code segment (e.g. C) until its parent segment (e.g. B) is committed and its sibling NT-Path (e.g. E) is squashed. Otherwise, its sibling NT-Path (e.g. E) may read data generated by this taken path segment (e.g. C), violating the data dependency described above. In other words, in order to commit, a taken path segment needs two tokens, a commit-token from its parent segment and a squash-token from its sibling NT-Path. In cases

when a taken path is forced to commit, for example, when one of its dirty lines is to be displaced from L1 to L2, related sibling NT-Paths are squashed immediately, so that the taken path will not be stalled.

4.4. State Inconsistency Fix on NT-Paths

As mentioned in Section 3.2, NT-Paths may introduce state inconsistency that may result in false positives and false negatives in bug detection. To address this problem, PathExpander performs some simple variable consistency fixes.

We should note that, 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 variables depend. Fortunately, such complete fix is also not required in PathExpander usage scenario as discussed in Section 3.2. Therefore, current PathExpander prototype only fixes the condition variables of the branch corresponding to a given NT-Path. We will investigate using program analysis, symbolic execution [8], or reverse execution with dynamic slice information [40], to perform more sophisticated consistency fix in our future work.

(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 true in the NT-Path by changing x's value to anything smaller than 5 to force executing the TRUE edge. To make the fix more accurate, one can rely on static analysis and value-invariants inference [12] to pick a value satisfying not only the desired branch direction but also the *n*ormal value range and usage pattern of this variable. In our current implementation, if only one value satisfies the condition (e.g. an equality condition), this variable is fixed to be this value at the entry of the NT-Path. If a range of values satisfy the condition (e.g. an inequality condition), the variable is fixed to be either exactly the boundary value, or close to the boundary.

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 PathExpander decides to execute the non-taken path as an NT-Path, the pointer p needs to point to some real data. Otherwise, it may crash or introduce false positives in bug detection. To address this problem, PathExpander 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, PathExpander 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 7.1).

(2) Who Fixes Key Variables? PathExpander uses a compiler to insert variable-fixing instructions at the beginning of each branch edge that could lead to an NT-Path. There are two ways to estimate whether a branch edge can lead to 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 possibly be spawned to NT-Paths. Therefore, variable-fixing instructions are inserted in both edges.

Line Number	Original Code	PathExpander Code
1	if (x≥2) {	if (x≥2) {
		x=2 $< p >$;
2	big(var,x);	big(var,x);
3	}else{	}else{
		x=1 $< p >$;
4	small(var,x);	small(var,x);
5	}	}

Table 1. Key variable fixing (x=2 <p>, x=1 <p> are predicated instructions executed only at the NT-Path entrances.)

(3) When 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, where the actual semantic is already followed. To provide the above functionality, variable-fixing instructions are *predicated* in a way similar to previous work [16, 23] so that these instructions are executed only at the entrance of an NT-Path. In all other cases, the predicate register is unset, and, consequently, these instructions behave like NOPs with little overhead.

(4) An Example

To illustrate our idea, Table 1 gives an example of how the variable-fixing is performed before executing an NT-Path. Suppose x's value is 0, and an NT-Path is spawned from line 2 (function big). To fix the condition variable x in this NT-Path, the compiler inserts a predicated instruction $x = 2 < p >$ to make the variable equal to 2, the boundary value. As a result, executing this path as an NT-Path is less inconsistent and thereby has smaller possibility for extra false positives (introduced by PathExpander) in bug detection. If this path is executed as a taken path, the predicate for 'x=2 $< p >$' is false. Therefore it behaves like a NOP. The process is similar for 'x=1 $< p >$' on the other edge.

5. Software-PathExpander Implementation

The pure software implementation of PathExpander is based on PIN [22], a dynamic binary code instrumentation tool. As a state-of-art instrumentation tool, PIN provides high-quality optimized instrumentation and many useful APIs, such as processor state checkpointing and register value modification.

NT-Path Selection We instrument every branch instructions to collect and maintain the dynamic branch exercise information. At run time, after each branch instruction is resolved, the instrumented analysis code dynamically makes decision whether an NT-Path should be spawned based on the exercise history maintained in a hash table.

NT-Path Spawn We use PIN API to first save the current processor states into a special *processor checkpoint* data structure, then directly modify the PC register to the non-taken branch edge, and proceed the execution to the NT-Path.

NT-Path Side Effects Sandbox. During the NT-Path execution, we instrument every memory write to log the old value of each overwritten memory location into a special *restore-log*, so that all memory effects can be rolled back when we want to resume the correct branch edge.

NT-Path Termination. The termination condition, i.e. length of NT-Path, unsafe NT-Path system call and NT-Path crash, is monitored by another set of instrumentation. When one of these three

termination conditions is satisfied, a software NT-Path squash and rollback routine is invoked. This routine writes all the overwritten memory values back to corresponding memory locations based on the *restore-log*, and then resets the processor registers based on the previous *processor checkpoint* record and continues the taken path.

6. Experimental Methodology

6.1. Simulator

To evaluate PathExpander, we augment an existing cycle-accurate execution-driven simulator [6, 41] that models a 4-core CMP with PathExpander functionality. The parameters of the architecture are shown in Table 2. We use one core when evaluating the standard configuration of PathExpander and use all 4 cores for the CMP optimization option.

Core Parameters			
Squash overhead	10 cycles	CPU frequency	2.4GHz
		Int, Mem, FP FUs	3, 2, 2
		BTB	2K, 2 way
Spawn overhead	20 cycles	ROB, I-window sizes	128, 64
		Fetch, Issue, Retire widths	6, 4, 4
		LD, ST queue entries	64, 48
L1 cache	16KB, 4-way, 32B/line 3 cycles latency (2 cycles for non-CMP)		
Memory System Parameters			
L2 cache	1MB, 8-way, 32B/line, 10 cycles latency		
Memory	200 cycles latency		

Table 2. Parameters of the simulation

6.2. Evaluated Dynamic Bug Detection Tools

To show the generality of PathExpander, we evaluate PathExpander using three different dynamic bug detection methods: software-only dynamic checker (CCured [27]), hardware-assisted dynamic checker (iWatcher [41]) and assertions. CCured is a publicly available, dynamic-static hybrid tool and we use its dynamic part. CCured and iWatcher can dynamically detect memory-related bugs. We use assertions to evaluate programs with semantic bugs.

It is easy to integrate these dynamic bug detectors with PathExpander. At run time, the checking code segments inserted by these detectors are automatically executed by hardware on both taken path and the NT-Paths. We just need to tag those checking functions in advance so that PathExpander does not spawn NT-Paths *within* them; and store those checking results to a special memory region so that they will not be discarded at the NT-Path rollback.

6.3. Evaluated Applications

We have conducted two sets of experiments. The first set uses *seven* buggy applications, containing 38 bugs in total, to evaluate the functionality of PathExpander in helping bug detection. The second adds three SPEC2000 benchmarks (gzip, vpr and parser) to evaluate the overhead of PathExpander, the performance benefits of CMP, etc.

The focus of our evaluation is to demonstrate how PathExpander can help dynamic bug detection tools to detect bugs in non-taken paths, not to compare these three bug detection methods. Therefore, we pick buggy applications with bugs that are possible to be detected by the dynamic detection methods. As CCured and iWatcher can detect only memory-related bugs, we use assertions for semantic bugs in the Siemens suite [13].

Application	LOC	#Bugs	Detection Tool
099.go	29,623	2	CCured and iWatcher
bc-1.06	17,042	2	
man-1.5h1	4,675	1	
Print_tokens	767	7	Assertions
Print_tokens2	727	10	
Schedule	411	8	
Schedule2	378	8	

Table 3. Applications and bugs evaluated

Table 3 gives the details about the 7 buggy applications. 099.go is from the SPEC95 benchmark, bc-1.06, man-1.5h1 are from the open-source community. They all contain memory-related bugs that fit for CCured and iWatcher. The other four are from the Siemens suite [13], which is widely used for evaluating software testing and bug detection methods. Each Siemens benchmark has several versions and each version has one semantic bug. Since the Print_tokens2 benchmark, version 10, contains a memory bug (Figure 1), we also use it to evaluate CCured and iWatcher.

Note that PathExpander only increases the path coverage of a dynamic checker, and does NOT help the checker to detect other types of bugs. Therefore, in table 3, the number of bugs here means the number of detectable bugs by the corresponding dynamic bug detection method in the last column, and do not include those types of bugs that cannot be detected by the corresponding tool.

To demonstrate the effectiveness of PathExpander in increasing the dynamic bug detection coverage, we use inputs that do *not* expose the tested bugs in normal execution. Almost all inputs used in our experiments are very general. If PathExpander can help expose bugs on NT-Path with these inputs, the probability of exposing the bugs to dynamic detection tools would be greatly increased, given a randomly generated test suit. Take Print_tokens2 version10 (Figure 1) as an example. The original bug triggering input file should contain a token that starts with quotation mark and does not have a second quotation mark. In contrast, the input we use *d*oes not have any constraint on the the starting character of the tokens in the file.

Additionally, we also evaluated the effectiveness of PathExpander with *multiple* different test cases based on the *cumulative* coverage increase. In particular, the Siemens benchmark suite provides many test cases for each benchmark and we randomly choose 50 cases for each application. 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 inputs, in addition to those provided by the *bc* package.

In our experiments, the threshold *MaxNTPathLength* is 100 instructions for the four small Siemens 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 have reached the end of the program before finishing 1000. The *NTPathCounterThreshold* is set to 5, and the *MaxNumNTPaths* for CMP-option is 32. All results are obtained using this default setup unless otherwise mentioned in Section 7.6, where we study the effects of these parameters.

We should note that, PathExpander is designed to improve path coverage. 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 PathExpander in increasing path coverage.

6.4. Software PathExpander Experiment Settings

In order to study the tradeoff between PathExpander hardware design and software implementation, we also run the software PathExpander on the same set of applications described earlier and compare the performance with our hardware design. The software PathExpander experiments are conducted on x86 machine with 2.4 GHz Pentium 4 processor, 1MB L2 cache and 1GB of memory.

7. Experimental Results

Dynamic Tools	Application	#Bug Tested	#Bug Detected	
			Baseline	PathExpander
Software Tool (CCured)	099.go	2	0	1
	bc-1.06	2	0	1
	man-1.5h1	1	0	1
	P_t2(v10)	1	0	1
Hardware Tool (iWatcher)	099.go	2	0	1
	bc-1.06	2	0	1
	man-1.5h1	1	0	1
	P_t2(v10)	1	0	1
Assertions	P_t	7	0	5
	P_t2	10	0	6
	Schedule	8	0	3
	Schedule2	8	0	4

Table 4. Bug detection results of PathExpander (Baseline means *no* PathExpander; P_t, P_t2 stand for Print_tokens and Print_tokens2.)

This section first shows the PathExpander's results in bug detection, false positives before and after consistency fix, and path coverage improvement with single and multiple inputs. All these results of different PathExpander implementation (i.e. software or hardware) are similar. We present the overhead results for the standard configuration and the CMP-optimization option; compare the tradeoff between the hardware and software implementations of PathExpander. Finally, we evaluate the effects of parameters setting.

7.1. Bug Detection Results

Benefits: As shown in Table 4, using common inputs that do not expose the tested bugs, the baseline case (without PathExpander) fails to detect any bug due to their path coverage limitation. However, with PathExpander, these methods can detect 21 of the 38 tested bugs using the same non-bug-triggering inputs used in the baseline case. For example, in the Print_tokens benchmark, PathExpander enables assertions to detect 5 out of 7 tested bugs. In the real application bc-1.06, PathExpander also helps CCured and iWatcher to detect one of the two tested bugs. Since the inputs used in our experiments are common and originally non-bug-triggering, the probability of exposing bugs to dynamic detection tools using random test generator is significantly improved by PathExpander. It saves programmers effort to design and generate special inputs to detect bugs in those uncovered paths.

Limitations: However, PathExpander is definitely not a panacea. PathExpander fails to help the tested dynamic bug detection tools detect 17 of the 38 tested bugs. The main causes and solutions to address each corresponding problem can be summarized as follows: (1) Some bugs such as the ones in Schedule version 1 and 3 cannot be detected by assertions with the help of PathExpander because they are limited by the value coverage problem instead of the path coverage problem. To address it would require a value-coverage related solution like the one by Austin et al [18]. (2) Some bug such as the one used in bc-1.06 is limited by the path-coverage problem, but the entry branch edge has been intensively exercised before the bug triggered. Therefore PathExpander would not explore it as an NT-Path. As a result, CCured and iWatcher still fail to detect the bug even with PathExpander. However, this problem can be addressed by adding random factor into PathExpander's NT-Path selection. (3) Some bugs such as the one in Print_tokens2 version 3 escape the detection due to the inconsistency problem introduced by the NT-Path execution. Addressing this issue needs more sophisticated consistency fixing techniques described in Section 4.4. (4) Some bugs such as the ones in Print_tokens2 version 6 and *go* can be detected by PathExpander if some special non-bug-triggering input is used. Since this input is as uncommon as the bug-triggering one, we do not count them as "being helped" by PathExpander.

7.2. Effects of Consistency Fixing

Table 5 shows the number of false positives and the bugs detected due to PathExpander, before and after consistency fixing with CCured and iWatcher. As false positives/negatives with the assertion method depend on how and where assertions are inserted, the results can be very subjective and not very meaningful. Therefore, we do not report the results with assertion.

Bug Detection Method	Application	#False Positives		#Bug Detected	
		Before	After	Before	After
Software Tool (CCured)	099.go	83	20	1	1
	bc-1.06	10	7	1	1
	man-1.5hl	7	0	0	1
	P_t2(v10)	4	1	1	1
Hardware Tool (iWatcher)	099.go	2	2	1	1
	bc-1.06	3	2	1	1
	man-1.5h1	0	0	0	1
	P_t2(v10)	0	0	1	1
Average		**13**	**4**	**0.75**	**1**

Table 5. False-positive pruning by key variable value fix (False positive results include only those caused by PathExpander, not those caused by the dynamic checker itself; P_t2 stands for Print_tokens2.)

Our results show that our consistency fixing techniques are effective in pruning false positives and detecting more bugs. Fixing key variables can help detect the bug in *man* and reduce the number of false positives from an average of 13 to only 4. Consistency fixing can help detect more bugs because some bugs in NT-Paths have

data dependencies on the branch condition variables and the inconsistency problem can interfere with the detection of these bugs.

However, some false positives still remain. For example, *go*'s remaining 20 false positives are caused by its intensive array operations and complicated variable correlations. These false positives can be further pruned by some extensions such as skipping initialization phases, more sophisticated consistency fixing, etc, which remain as our future work.

7.3. Coverage Improvement

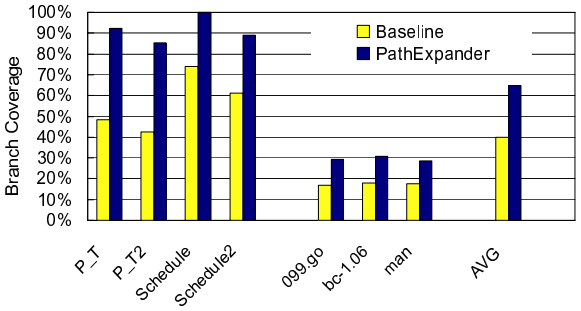

Figure 7. PathExpander's improvement on branch coverage (Baseline means *no* PathExpander.)

The main reason for the capability of PathExpander in helping detecting bugs with non-bug-triggering inputs is that PathExpander extends the detection coverage to include non-taken paths in the monitored run. Here, we show the quantitative measurement result of the coverage increase by PathExpander. Of course, the coverage results would better be interpreted slightly different from those in software testing, because there exists some variable inconsistency. We only use these results for illustration purposes.

As shown in Figure 7, PathExpander improves the branch coverage of the baseline by 35–100%. For example, in Print_tokens, the original monitored execution only exercises 48.3% branch edges of the whole program. With PathExpander, this percentage increases to 92.2%. Therefore, bugs 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, PathExpander increases branch coverage from 39.8% to 65.0%.

To understand more on PathExpander's effects on code coverage improvement, we also conduct an evaluation using *multiple different* test cases.

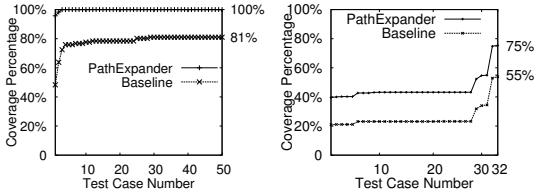

Figure 8. Cumulative branch coverage with multiple test cases

Figure 8 (a) gives the evaluation result with Print_tokens, a Siemens benchmark. The 50 inputs used here are randomly picked from the input pool provided by the Siemens benchmark. As we can see, after 50 inputs, Print_tokens' branch coverage becomes stable at around 80% in the baseline. As explained in Section 2, usually the last 5-20% branch edges are prohibitively hard to cover [33], but, fortunately, PathExpander can help to push bug detection into these remaining edges to achieve almost 100% coverage as shown in the figure. The results for other Siemens benchmarks are similar. Their result figures can be found in our technical report [21].

Testing in larger real world software is more difficult. Figure 8(b) shows such an example of *bc*, a moderate sized open source application. We use a production-rule based test generator to generate many random test inputs (the first 25 cases in the figure) in addition to the inputs provided by the *bc* programmers in the *bc* package. The final cumulative branch coverage without PathExpander is around 55%, and PathExpander improves that to 75.4%.

7.4. Hardware PathExpander Overhead

In order to measure the execution performance, we use 3 SPEC benchmarks and 3 real applications (bc, man, go). We do not use the Siemens benchmarks here because they are too small to show meaningful performance results. As shown in Figure 9, with the standard configuration, PathExpander imposes an average of 50.4% overhead for tested applications. The overhead with the SPEC benchmarks are relatively small, only 4.5-23.2%, because they have lower NT-Paths spawning frequency than the open-source real applications (Table 6). With the CMP optimization, PathExpander's overhead is significantly minimized to only less than 10% for all applications. The main reason for such a low overhead is that the NT-Paths are executed on idle cores concurrently with taken paths.

Our results show that hardware PathExpander is definitely a more efficient and cheaper way to improve the dynamic bug detection coverage than conventional methods, which usually require significantly more runs with different inputs. Taking application *bc* as example, with only 64.3% overhead (9.9% with CMP optimization), PathExpander explores 623 new paths and increases the branch coverage by 76%. Similar improvement may need 623 different runs with dedicatedly generated test inputs, which means 623 *times* overhead excluding the test generation overhead.

	#Spawns per MIns	Avg. NT-Ins per MIns	#Predicated Ins per MIns	L1 Miss (%)		
				Baseline	PathExpander	CMP-Opt
164.gzip	6.0	59K	106K	5.52	6.39	5.55
175.vpr	9.3	9K	103K	3.64	3.66	3.66
195.parser	23.1	22K	136K	1.96	2.02	1.97
099.go	364.3	375K	75K	2.32	4.43	2.35
bc-1.06	669.3	372K	45K	0.39	3.63	0.46
man-1.5h1	821.9	688K	41K	1.47	4.03	1.53

Table 6. PathExpander detailed performance results (MIns denotes one million instructions counting both NT-Path and taken-path instructions. By default, PathExpander means our standard configuration unless specified by CMP-Opt.)

Table 6 shows the detailed performance results with the hardware PathExpander. The overhead of the standard PathExpander comes from three main sources: NT-Path spawning and termination; NT-Path execution; and slightly higher cache miss rate. As for the CMP option, the optimization effects vary. On one hand, it hides NT-Path execution overhead by parallelism. Therefore, for applications such as gzip, bc, man and go, CMP-optimization successfully decreases their overhead from 23.2%–142.3% to less than

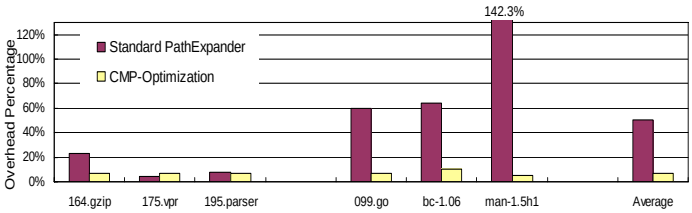

Figure 9. PathExpander (hardware design) overhead

Slow Down (times)	164.gzip	175.vpr	195.parser	Average
	385.8X	373.2X	393.4X	1240.7X
	0.99go	bc-1.06	man-1.5h1	
	1644X	2137X	2511X	

Figure 10. PathExpander software implementation overhead (The overhead of PIN dynamic code rewriting is NOT included.)

9.9%. On the other hand, it also introduces extra overhead, such as worse cache performance due to cache versioning, extra register copy between different cores, etc. If the original sequential overhead is small, these extra overheads may not be totally hidden by the parallelism benefit. That is why for parser and vpr, the optimization effect is less pronounced and even negative.

7.5. Comparison between PathExpander Hardware and Software Implementations

Figure 10 shows the performance of PathExpander's software implementation on the same set of applications and workloads. As we can see, the slow down ranges from 373 times to 2511 times, and is 1240.7 times on average. Among different applications, the performance distribution shares the similar trend as that in our previous hardware PathExpander experiments.

The large overhead comes from almost every modules of software-PathExpander: 1) NT-Path spawning needs software checkpointing, which is very expensive and contributes a big part to the large slow down; 2) logging-based sandboxing and rollback introduce a lot of extra memory writes; 3) instrumentation-based branch exercise history maintenance adds extra overhead to the execution of taken-path.

Comparing the PathExpander hardware and software implementations, they differ with each other on the hardware cost and big performance gap. Our hardware implementations incur 3–4 orders of *magnitude* smaller overhead than the software implementation, which means significant reduction in the programmers' debugging time! Such gain is achieved via simple hardware extensions including the hardware checkpointing and sandboxing which are already available in the emerging transaction memory architecture [2, 11, 32], and a simple extension to the BTB to add two exercise counters per entry. *More importantly, the hardware implementation also provides an interesting and promising opportunity to exploit idle cores on multicore architectures to enhance software quality*. All these reasons validate that hardware PathExpander is definitely worthwhile for exploration.

As for the software-PathExpander, though much slower than its hardware counterpart, it still provides a useful option to increase coverage of dynamic bug detection on existing machines to detect bugs on those paths that are difficult to generate test cases to cover.

7.6. Effects of Parameters

We also evaluate the performance effects of the two parameters, namely *MaxNTPathLength* and *MaxNumNTPaths*. The result figures can be found in our technical report [21]. Our results show that, with the increase of these two parameters, overhead increases gradually in the standard configuration, but only slightly in CMP optimization, because the latter can hide the extra NT-Path spawning and exploration by parallel execution. The CMP optimization provides PathExpander good potential to conduct more intensive NT-Path exploration, hence more help to dynamic bug detection.

8. Related Work

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

Multipath Branch Execution Several previous studies [1, 3, 15, 17, 36] 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 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 we want to detect bugs. In contrast, the multipath architectures need to run multiple branch edges only until the branch is resolved. Therefore, PathExpander needs to buffer many more side effects than the above works. (2) PathExpander 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) PathExpander needs to address the state inconsistency problem because it may introduce false positives and negatives in bug detection.

Value Coverage Problem A recent work conducted by Larson and Austin addressed a closely related, complementary, but *different* coverage problem called *value coverage* (value ranges of variables) for detecting buffer or string overflows [18]. 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. This work is very useful in increasing the value coverage for buffer overflow detection, but does not address the path coverage problem. In contrast, our work exactly addresses this path coverage problem and therefore well complements theirs.

Dynamic Bug Detection and Debugging Many tools have been proposed for dynamic execution monitoring. Well-known examples include Purify [14], Valgrind [28], CCured [27] and others [4, 20]. PathExpander would benefit almost all such tools by increasing their path coverage in each monitored run.

Delta debugging[39] changes variable values on-the-fly to force the control flow to different paths in order to find a correct execution that is very similar to a known wrong one for postmortem

bug diagnoses. PathExpander also forces control flow down different paths, but for a very different goal: increasing the coverage for dynamic bug detection tools. As such, PathExpander needs to address very different design issues such as how to sandbox the side-effects of non-taken paths, how to select which non-taken paths to explore, etc. In addition, PathExpander leverages hardware support and thereby is much more efficient.

Recently, many researchers have shown the effectiveness of architectural support for software debugging. Just to name a few, ReEnact [31], Flight Data Recorder [38], BugNet [26], and many others [35, 41] are examples of architectural innovations to improve software robustness. None of them addresses the path coverage problem of commonly-used dynamic bug detection methods, and PathExpander can benefit them in detecting bugs in non-taken paths during each single monitored run.

Model Checking and Static Analysis Model checking [25] 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 PathExpander is a dynamic approach and requires no specification or annotation.

9. Conclusions

This paper addresses the fundamental problem of path coverage for commonly-used dynamic bug detection tools. Specifically, we have proposed an innovative, automatic, general and low-overhead approach, called PathExpander, which dynamically increases the path coverage of dynamic bug detection and thus reduces the amount of efforts required for software engineers to design and implement test cases to cover these extra paths in testing. We have also presented two implementations of PathExpander, one with simple hardware extensions and the other purely in software.

Our experiments with seven buggy programs and three different dynamic bug detection methods using general inputs that do not expose the tested bugs show that PathExpander is able to help these tools detect 21 out of 38 tested bugs that would otherwise be missed. This is because PathExpander increases the branch coverage of the monitored execution from 40% to 65% on average. In addition, it incurs few (4 on average) false positives with simple consistency fixes. Our results also show that the hardware PathExpander imposes small overhead (less than 9.9% with CMP option), 3–4 orders of magnitude lower than the pure-software implementation.

We believe that our work provides a fundamentally different way to address the path coverage problem in dynamic bug detection and thereby well compliments other research work in bug detection and software testing. In addition, our work also provides a strong demonstration case of leveraging hardware, particularly the multi-core architecture and the sandbox/rollback functionality available in the emerging transaction memory architecture, for software engineer tasks. As hardware vendors are searching for innovations to enrich the microprocessor capability in areas in addition to performance, our work provides one of the first studies on improving the efficiency of general software testing, a challenging and important research problem that is attracting much attention, and thereby will likely inspire many follow-up works in this promising direction.

References

[1] P. S. Ahuja, K. Skadron, M. Martonosi, and D. W. Clark. Multipath execution: Opportunities and limits. In *ICS*, 1998.

[2] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA*, 2005.

[3] J. Aragn, J. Gonzlez, A. Gonzlez, and J. Smith. Dual path instruction processing. In *ICS*, 2002.

[4] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *PLDI*, June 1994.

[5] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. In *SPIN*, 2005.

[6] M. Cintra, J. Martinez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory systems. In *ISCA*, June 2000.

[7] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP*, October 2003.

[8] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI*, 2005.

[9] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang. Characterization of linux kernel behavior under errors. In *DSN*, 2003.

[10] N. Gupta, A. P. Mathur, and M. L. Soffa. Automated test data generation using an iterative relaxation method. In *FSE*, 1998.

[11] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA*, 2004.

[12] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE*, May 2002.

[13] M. J. Harrold and G. Rothermel. Siemens Programs, HR Variants. URL: http://www.cc.gatech.edu/aristotle/Tools/subjects/.

[14] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Usenix Technical Conference*, 1992.

[15] T. Heil and J. Smith. Selective dual path execution. In *Technical report, University of Wisconsin - Madison*, 1996.

[16] P. T. Hsu and E. Davidson. Highly concurrent scalar processing. In *ISCA*, 1986.

[17] A. Klauser, A. Paithankar, and D. Grunwald. Selective eager execution on the polypath architecture. In *ISCA*, 1998.

[18] E. Larson and T. Austin. High coverage detection of input-related security faults. In *USENIX Security Symposium*, August 2003.

[19] J. Larus. Building dependable software. *ASPLOS XI*, 2004.

[20] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.

[21] S. Lu, P. Zhou, W. Liu, Y. Zhou, and J. Torrellas. Spancoverage: Architectural support for increasing the path coverage of dynamic bug detection. Technical Report UIUC-DCS-R-2005-2618, Univ. of Illinois, 2005.

[22] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[23] S. A. Mahlke et al. A comparison of full and partial predicated execution support for ILP processors. In *ISCA*, 1995.

[24] J. F. Martinez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas. Cherry: checkpointed early resource recycling in out-of-order microprocessors. In *MICRO*, 2002.

[25] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *OSDI*, 2002.

[26] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ISCA*, 2005.

[27] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *POPL*, January 2002.

[28] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *ENTCS*, 2003.

[29] J. Oplinger and M. S. Lam. Enhancing software reliability with speculative threads. In *ASPLOS*, October 2002.

[30] M. Prvulovic, M. J. Garzarán, L. Rauchwerger, and J. Torrellas. Removing architectural bottlenecks to the scalability of speculative parallelization. In *ISCA*, 2001.

[31] M. Prvulovic and J. Torrellas. ReEnact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *ISCA*, June 2003.

[32] R. Rajwar and J. R. Goodman. Transactional execution: Toward reliable, high-performance multithreading. In *Micro*, 2003.

[33] Roper and Marc. *Software Testing*. McGrawHill, 1994.

[34] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE*, 2005.

[35] N. Tuck, B. Calder, and G. Varghese. Hardware and binary modification support for code pointer protection from buffer overflow. In *Micro*, 2004.

[36] S. Wallace, B. Calder, and D. M. Tullsen. Threaded multiple path execution. In *ISCA*, 1998.

[37] N. Wang, M. Fertig, and S. Patel. Y-branches: When you come to a fork in the road, take it. In *PACT*, 2003.

[38] M. Xu, R. Bodik, and M. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA*, 2003.

[39] A. Zeller. Isolating cause-effect chains from computer programs. In *SIGSOFT FSE*, 2002.

[40] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *ICSE*, 2003.

[41] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient Architecture Support for Software Debugging. In *ISCA*, 2004.