Software Profiling for Deterministic Replay Debugging of User Code

Satish NARAYANASAMY, Cristiano PEREIRA and Brad CALDER
Department of Computer Science and Engineering
University of California, San Diego
{satish, cpereira, calder}@cs.ucsd.edu

Abstract.
Significant time is spent by companies in trying to reproduce and fix bugs in their software. The process of testing and debugging can immensely benefit from a tool that supports Deterministic Replay Debugging (DRD). A tool that supports DRD will allow a user to record a program's execution in a log, and to deterministically replay every single instruction executed as part of the application using the log.

In this paper, we present the BugNet software tool which can support DRD. It can handle all forms of non-determinism, including the non-determinism due to thread interactions in a multi-threaded application. Since the BugNet tool is operating system independent, a program execution that fails in a particular user's environment can be easily captured by the user in a log, and later reproduced by a developer working in a completely different environment.

We also empirically quantify certain variables relevant to the DRD process. This includes an empirical analysis, that quantifies how much of a program execution has to be logged and replayed in order to understand the root cause of a bug. Further, we examine the potential benefit of using dynamic slicing along with our deterministic replay debugger.

Keywords. Debugging, Deterministic Replay, BugNet

1. Introduction

Tracking down and fixing bugs in software can be a nightmare, costing a significant amount of time and money. These software bugs account for nearly 40% of computer system failures [15] and according to NIST [19] they cost the U.S. economy an estimated $59.5 billion annually.

When a program execution fails due to a software bug, the programmer should have the ability to *deterministically* replay the execution as many times as needed, in order to understand and fix the bug. By deterministic replay, we mean the ability to re-execute exactly the same sequence of instructions with exactly the same input like in the original execution that exposed the bug. We call this method of debugging as *Deterministic Replay Debugging*(DRD). DRD enables the programmer to go back to any instant during the program execution and start replaying from thereon to understand the bug.

We recently proposed a hardware logging mechanism called BugNet [17] to support building DRD tools. In this paper, we adapt some of the techniques introduced in our hardware logging mechanism to build a BugNet profiling tool completely in software without assuming any hardware support. The BugNet profiling tool can continuously log information pertaining to the program execution, which can be used by a developer to deterministically replay the program. One important feature of our mechanism is that it can support deterministic replay of program execution in the presence of all forms of non-determinism, including non-determinism due to system interactions (eg: I/O), and also thread interactions in multi-threaded applications. The ability to deterministically replay a program execution is vital to reproduce non-deterministic bugs.

A majority of application level bugs can be debugged by just replaying the instructions executed in the user code and the shared libraries. Therefore, our BugNet mechanism focuses on deterministically replaying only the instructions executed in the user code and the shared libraries. While our BugNet tool can profile user mode execution across various system events, it does not attempt to profile the instructions executed as part of the system in the kernel mode. To enable deterministic replay of only the user code and the shared libraries, BugNet logs the values of the load instructions executed by the application. This logging approach is different from the copy-on-write checkpointing technique used in the earlier works [7,25,8,12]. Along with a copy-on-write checkpoint technique, these earlier systems explicitly logged I/O events and other interrupts in order to support deterministic replay of the application across system events, which restricted their tools to be dependent on the operating system and also added significant complexity in developing and maintaining such tools. Unlike these tools, BugNet's load value logging approach makes it completely operating system independent, which makes it easier to develop and maintain. Because of the same reason, a program execution that fails in a particular user's environment can be captured by the user in a log using BugNet, and easily reproduced by a developer working in a completely different environment.

In this paper, we present a software version of BugNet [17], that can be used by the developers and quality assurance engineers to efficiently track down bugs. We have implemented our logger and replayer using the Pin dynamic instrumentation tool [14]. The contribution of this paper over our hardware proposal [17] is that the prior work focused on the hardware support for logging with very little performance overhead, and not an efficient software implementation. A naive software version of BugNet is prone to incur a heavy performance overhead. In this paper, we discuss a software implementation that mitigates the performance issues. The previous hardware approach logged shared memory dependencies for multi-threaded programs by observing the coherence messages in hardware. Here, in this paper, we describe a software logging algorithm to capture shared memory dependencies in order to deterministically replay multi-threaded programs.

We empirically quantify two metrics relevant to Deterministic Replay Debugging by analyzing the bugs in some of the popular open source programs and in the Siemens [11] benchmark suite. We improve the analysis presented in our prior work [17] to quantify what we call the *replay window* length, which is the length of program execution (in terms of instructions) that need to be logged and replayed in order to understand and fix a bug. We also study the size of the dynamic slice [27,30] of a replay window. The first metric corresponds to how much state needs to be logged in order to capture the bug. The second metric corresponds to how much state a developer may need to examine in order to track down and fix the bug.

2. Related Work

There have been a few debugging tools proposed in the past which can enable replay of program execution. Before discussing these tools, we first summarize the checkpoint mechanisms used in these tools, which is central to any tool that can support replay.

2.1. Checkpoint Schemes

At the core of any system that enables replaying program execution is a checkpoint mechanism. A checkpoint is created at regular intervals of program execution and these intervals are usually referred to as checkpoint intervals. At the beginning of a checkpoint interval, a snapshot of the program's execution state is logged. This snapshot includes information such as CPU registers, memory, and also kernel states corresponding to the process such as file descriptor tables, signal handlers, etc. A commonly employed technique to reduce the log size is the copy-on-write policy. Instead of logging all the state values at the beginning of a checkpoint interval, a state value is logged only when the state gets modified for the first time within the checkpoint interval. The state of the process at the beginning of a checkpoint interval can be rebuilt, during replay, by starting with the current process's state and restoring the values from the log. Henceforth, we will refer to a log collected using this checkpoint mechanism as an *undo-log* [12].

In addition to the *undo-log*, the above checkpoint mechanism also requires that sufficient information related to the occurrence of non-deterministic events such as interrupts, OS calls, DMA transfers and also data races in multi-threaded programs are logged. This log, which is referred to as *redo-log* [12], is required to reproduce these non-deterministic events to achieve deterministic replay.

2.2. Replay Debugging

We will now discuss systems that enable logging and replaying a program execution. All of these system employ some form of copy-on-write checkpoint mechanism.

Many previously proposed systems [8,5,4,1] focus on replaying programs that have only deterministic system interactions. Hence they do not have the complexity of logging the non-determinism during program execution. These systems cannot aid in debugging programs whose execution are affected by non-determinism due to system interactions or races in multi-threaded programs.

Bugs due to data races in multi-threaded programs are difficult to debug and there have be studies to statically detect data races [22]. There have also been some work in building deterministic replayers to support debugging multi-threaded applications. InstantReplay [13] is a software based deterministic replayer that records the memory access order, and a hardware-assisted version was presented by Bacon and Goldstein [2]. Netzer [18] proposed an optimization to reduce the amount of shared memory dependency information that needs to be logged. If one assumes that the multi-threaded application is running on a single processor, then one can limit the amount of recording to just the scheduler decisions [6,20]. The above schemes tackle the non-determinism specifically for data races in multi-threaded programs, but they do not focus on handling other forms of non-determinism such as I/O interactions.

Flashback [25] provides lightweight operating system support for fine grained rollback by implementing a copy-on-write checkpoint scheme. It can also support replaying of the program's execution. It can handle non-determinism due I/O interactions, such as input and output from file operations by collecting logs similar to the redo logs that we described earlier. However, it has a limitation in that, it cannot handle asynchronous I/O. For example, I/O interactions with external devices such as network drivers cannot be deterministically replayed. This is because in their replay system, the OS cannot control the state of the external devices and induce the exact same behavior in them.

ReVirt [7] and TTVM [12] uses virtual machine support to enable deterministic replay. They both support deterministic replay of I/O interactions, by creating *redo* logs that contain input information from I/O calls and timestamps for the interrupts. While the goal of ReVirt was to enable detailed intrusion analysis by supporting deterministic replay, TTVM focused on debugging just the operating system code. ReVirt does not provide support for deterministically replaying multi-threaded applications on a chip multi-processor.

Flight Data Recorder [29] uses hardware support to enable deterministic replay debugging. It uses a copy-on-write hardware checkpointing scheme to create an undo-log. The FDR hardware also collects a redo-log to capture I/O and interrupts explicitly. Using the logs one can deterministic replay the fully system.

Unlike all of the above mechanisms, the BugNet software profiling tool employs a different type of checkpoint mechanism that is based on logging the output of load values. Our load-based checkpoint mechanism is operating system independent and hence it is relatively easy to develop and maintain. Also, it naturally handles I/O and other such system interactions. Further, our tool can deterministically replay multi-threaded programs as well. We describe software algorithms to implement the BugNet logger and replayer using a dynamic instrumentation tool.

In parallel with our development of BugNet, Bhansali et. al. [3] have developed a tool to support Deterministic Replay Debugging using a load-based checkpointing scheme. However, they did not focus on precisely capturing the shared memory dependencies, which we can capture using our BugNet software tool.

3. BugNet Overview

In this section we provide an overview of the BugNet checkpointing approach and describe its ability to rollback execution and support deterministic replay debugging.

3.1. BugNet Overview

Our software implementation of BugNet consists of both a logger and replay debugger provided as two binary instrumentation tools, built using Pin [14]. Pin is a dynamic instrumentation infrastructure, which uses dynamic compilation to instrument binaries dynamically as the program executes. It allows users to instrument specific functions and instructions to embed call-backs at these places to the analysis routines. The analysis routines can examine, log and replay the execution of an instrumented program.

BugNet is built on the observation that a program's execution is essentially driven by the values read when executing load instructions. We can therefore deterministically replay a program's execution starting from an arbitrary point if we are given the starting PC and register state at that point in execution, and all of the load values from that point on until the end of execution. This information is what we refer to as a checkpoint. To maintain these checkpoints we break a program's execution into checkpoint intervals. A *checkpoint interval* represents a window of committed instructions that is captured by the checkpoint being logged.

In BugNet, a new checkpoint is created at the beginning of each checkpoint interval to allow execution to be replayed starting at the first instruction of the interval. When using the BugNet logger, a maximum size (number of committed instructions) is specified for the checkpoint interval. When this limit is reached, a new checkpoint interval is initiated. For a checkpoint interval, enough information is recorded in a log to start the replay of the program's execution at the start of that checkpoint interval.

At the start of a checkpoint interval, a snapshot of the architectural state is recorded. The recorded architectural state includes the program counter and register values. After initialization, whenever a load instruction is executed the address of the load and the value that is loaded is written into the checkpoint log.

However, recording the result value of every load instruction would clearly be expensive. One of the optimizations in the BugNet logger is that a load value is recorded only if it is the first access to the memory location in the checkpoint interval or if the memory location has been modified by another thread or due to any system effects. The result value of the other loads can be trivially and deterministically regenerated during replay. In order to determine which load values need to be logged, our software BugNet logger keeps track of the values of each memory location that has already been logged for the current checkpoint interval in a profiling data structure. While executing a load, if the memory address has already been logged, and if the value in the data structure is same as the value in the application's memory (which will be true only if the memory location has not been modified by the system or a remote thread), then the logger does not log the load value. Otherwise, it logs the load value and the data structure is updated to keep track of the address and value so that, in future, it can avoid logging when a load accesses the same memory location. The details about this optimization are described in Section 4.

In the case of multi-threaded programs, if we log all the load values for a thread, then it can enable us to replay each thread individually. Even when we apply the above optimization, since we make sure to log the memory locations modified by the remote threads, we can still replay each thread individually. However, in order to debug a multi-threaded program, we need to additionally record shared memory dependencies across the threads. If we can precisely capture these dependencies, then a replayer using these logs can support powerful debugging features such as single stepping through the multi-threaded program execution in both directions. We explain an algorithm to achieve this in Section 5.

3.2. Ability to Replay a Checkpoint

Our checkpointing approach allows a user to instantly start execution at any given checkpoint for debugging. The main steps consist of setting the PC to the logged PC and restoring the logged register state. We can then start to deterministically re-execute the instructions one at a time, and each load value that was logged will get its value from the checkpoint.

3.3. Focus on Debugging Only User Code

We limit our focus on supporting deterministic replay for only the user code and the dynamic shared libraries. This provides the ability to debug application level bugs. Note, we can still replay the application across the system events. The main benefit of limiting ourselves to focus just on application level replay is that it simplifies our logger and replayer implementation. This is because, we do not have to trace the kernel mode execution, and by using our load value based checkpoint approach, we can easily capture a program's execution across interrupts and system calls, and replay it. Also, our implementation and our approach is almost completely independent of the operating system. This is an advantage, because we would not have to release new versions of our tool when there are operating system patches or updates. Independence from the system environment also enables co-operative development among users working in different environments. For example, a developer can replay a program execution that was traced in the customer's environment.

3.4. BugNet Use Cases

Our BugNet software tool is useful during development and testing to capture and reproduce the bugs, including the notorious non-deterministic bugs. The program under test is executed under the BugNet logger, which uses Pin [14] to dynamically instrument the binary and produces the logs. If the program crashes, or if the tester observes an incorrect program behavior, the generated logs can be used to repeatedly replay the program deterministically, as many times as required to characterize and fix the bug. The program execution can be replayed from the beginning of any checkpoint interval. Based on this replay mechanism, one can easily build an interactive debugger that allows one to go backwards in time [5,8].

In addition to the above uses related to fixing a bug, the ability to deterministic replay a program's execution also allows one to perform more sophisticated dynamic analysis *offline* over that program execution to automatically detect bugs [23,9,21]. Many of these dynamic analysis tools are limited in their use due to their runtime overhead. Especially, programs with significant user interaction cannot be analyzed, since the dynamic analysis results in intolerable performance overhead. However, using a tool like BugNet profiler, one can collect just the necessary information to replay a program execution and use the collected logs to repeatedly replay and perform many different time consuming dynamic analysis to detect bugs such as memory access bugs [23,9] and data race bugs [21].

4. BugNet Logger and Replayer

In this section we will describe the implementation of our logger and replayer. The logger is used to collect checkpoint logs during testing for an input. On observing a bug, the collected checkpoint logs can be used by the developer to deterministically replay and debug the program's execution leading up to the bug.

4.1. Logger

The logger is implemented using the Pin dynamic instrumentation tool [14]. The purpose of the logger is to collect enough information about the execution of the program that can then be used to deterministically replay the program execution.

When running the logger, a user can choose a checkpoint interval length, which determines a limit on the total number of instructions executed in a checkpoint interval. A new checkpoint is created if the number of instructions executed in the current checkpoint interval reaches the limit. Each checkpoint interval can be replayed completely by itself. During replay, the replayer can start executing from the beginning of any of the checkpoints that were collected, as well as work their way backward through execution. In the case of a multi-threaded program, there exists a separate checkpoint log file for each thread that is executed. We will later describe in Section 5 how to capture shared memory dependencies for multi-threaded programs so that we can deterministic replay them.

4.1.1. Checkpoint Header

We use Pin [14] to instrument the binary being examined at the basic block granularity to keep track of the number of instructions that are executed. When the number of instructions executed in a checkpoint interval reaches the checkpoint interval limit, we start a new checkpoint.

Figure 1 illustrates the working of our software BugNet logger. We will use this as an example to explain our logger implementation. The vertical arrow in the figure represents the time of program execution. The two shaded boxes represent the creation of two checkpoints, and the instructions executed between these two checkpoints belong to a checkpoint interval.

For a new checkpoint the following initial header information is logged:

Process ID and Thread ID - are required to associate the checkpoint log with the thread of execution for which it was created.

Program Counter and Register File contents - are needed to represent the architectural state at the beginning of the checkpoint interval. This information will later be used by the replayer to initialize the architectural states before replaying the program execution using the recorded load values.

Time stamp - is used to synchronize between multiple threads in a multi-threaded program. This time stamp is nothing but a global counter value that represents the total number of checkpoints created at any instant of time.

4.1.2. Logging of Load Values and System Effects

In order to replay a program's execution from the beginning of a checkpoint interval all that we need to log are the results of the load instructions. Hence we instrument all of the load instructions to capture their output values and some of these are stored as part of the checkpoint log. We also need to instrument every store so that we know what the memory state of each thread is during its execution. This is required in order to capture the system effects. Logging the output of all the load instructions will clearly be expensive in terms of the log size. To optimize the log size, we make use of a simple observation that only the first value of each memory address loaded in a checkpoint needs to be logged. For example, in the Figure 1, we log $Load1$ as it is the first memory access to the location A. However, the following $Load2$ to the same address is not logged. For the memory location B, $Store1$ accesses it for the first time in the checkpoint interval. Since all the store values can be reproduced during the replay, we do not have to log their values. We also do not log $Load3$, even-though it is the first load access to the memory location B. This is because, during replay we know the value in the location B as we can replay the preceding $Store1$ to this location in the same checkpoint interval. Note that all the load/store addresses can be recalculated during the re-execution of the checkpoint, so they do not need to be logged.

In order to determine if a load is a first load to a memory location, we maintain a cache structure called *UserMemCache* per thread. This cache structure contains the values loaded from memory as seen by the application. The *UserMemCache* is an efficient data structure as it exploits the temporal and spatial locality of program accesses. When executing a load to an address Addr, if there is a miss in the *UserMemCache*, it is either the first time a load to Addr is executed, or the cache entry containing the value for the address Addr was kicked out due to a conflict. In either case, we log the value of the load.

If there is a hit in UserMemCache but the value in the application's memory differs from the value in the *UserMemCache* structure, then it means that the value was changed by some other entity other than the user thread. This entity could be another thread sharing the same address or the operating system. We therefore need to log this new value. This is an important difference from our prior hardware approach in [17]. The prior approach had to restart a checkpoint every time a system call occurred, because it could not keep a per thread shadow copy of the recently used values, which is effectively what the *UserMemCache* is. Unlike the hardware approach, here we do not have to restart a new checkpoint on encountering a system call or an interrupt.

Whenever a load for the same address is executed, its address and value are looked up in the structure. If there is a miss, we log the value and insert the load into the cache. If there is a hit, we

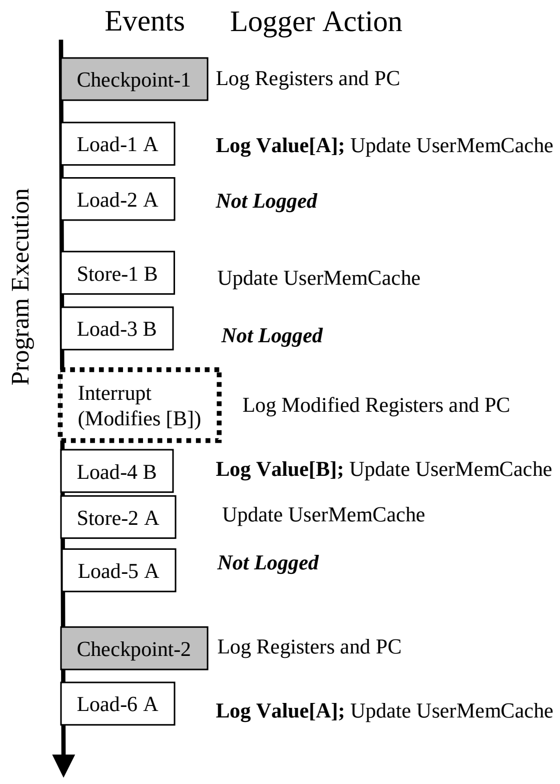

Figure 1. Example illustrating the working of the BugNet software logger.

compare the value in the cache with the actual value in memory. If the values mismatch, we again log the value because the memory state as maintained in *UserMemCache* is not up to date with application's memory because the application's memory should have been modified by an external entity as described above. The *UserMemCache* will be up to date with application's memory as long as it is modified only by the application through store instructions. To achieve this, we instrument every store and make sure that we update *UserMemCache* with the store's output values.

For example, in the Figure 1, when an interrupt occurs, we do not restart a new checkpoint. In this example, the interrupt modifies the location B. When the program executes $Load4$ to the location B, the logger compares the value in the *UserMemCache* with the value in the program's memory. Since the two values mismatch, the logger correctly logs the current value in the program's memory. Thereby, it correctly logs all the modifications done by the system. Note, the value of the $Load5$ to the location A is not logged as the value for the location A has already been logged, and also because it has not been modified by the interrupt. We can determine the second condition only because we correctly keep track of the values updated by the application through the stores (in this example, $Store2$).

The above load logging technique will be especially effective for long checkpoint intervals. This is because the greater the number of loads executed, the higher the probability that a memory location has already been logged. As a result, the amount of information recorded to replay an instruction will decrease with longer checkpoint intervals.

4.1.3. Reducing Log Size with a Hash Table

The *UserMemCache* is an efficient data structure as it exploits the temporal and spatial locality of program accesses. However, whenever a new block of memory needs to be brought in, an old block needs to be evicted. If the evicted block is accessed in the future, its values are redundantly logged. In order to reduce the log sizes, we use a hash structure called *UserMemHash* to back up the values that get evicted from *UserMemCache*. The UserMemHash is a global structure shared among all threads, whereas there is a *UserMemCache* per thread during logging.

At the beginning of a checkpoint interval, like the *UserMemCache*, the *UserMemHash* is also cleared. When a load misses in the *UserMemCache*, we look it up in the *UserMemHash*. If the load address is found then in the *UserMemHash*, then we use it to determine if we need to log the load value or not. If the load address is not found in the *UserMemHash*, then we assume the value for that address location is zero. This is valid, because during replay, we initialize all the memory locations to zeroes as well. If the value in *UserMemHash* is same as the value in the application's memory, then we do not generate a log entry for the load. Otherwise, we generate a log entry. The values in the *UserMemHash* are used to update the *UserMemCache* for the cache line corresponding to the load's address. Every time a cache line is evicted from *UserMemCache*, we store it in the *UserMemHash* structure. Compared to BugNet hardware support [17], it eliminates those logs that are redundantly logged due to cache capacity constraints. The pseudo-code in Figure 4.1.3 summarizes the logging algorithm.

We found that we benefited from using the *UserMemCache* with the *UserMemHash* structure because for some programs the logging overhead is much lower by using a *UserMemCache* structure. This is because we only access the *UserMemHash* when we get a miss in *UserMemCache*. Part of the overhead of the *UserMemHash* comes from the fact that if it grows too large, the logger can start having a lot of capacity missing. In addition, there is additional overhead for shared memory multi-threaded programs from having to use a lock when accessing the *UserMemHash*, whereas accesses to the per thread *UserMemCache* requires no locking during logging.

Note, the benefit of the *UserMemHash* is that it can significantly reduce the log size, but at the same time if it is not managed correctly during logging it can significantly slow down the program's execution for the above reasons. Therefore, during logging we monitor the size of this performance sensitive *UserMemHash* structure and make sure that the structure is cleared when it exceeds a critical size to avoid it from adversely impacting the performance of the logger.

4.1.4. Format of Load Value Record

Each log entry in the checkpoint for a logged load contains the effective address of the load instruction and the result value of the load instruction. The following is the format of a log entry:

```
(SV-Type, Reduced / Full Stride-Value,
 LV-Type, Reduced / Full Load-Value )
```

The first two fields are used to log the load stride value. The load stride represents the number of load operations that should be executed before restoring the next load value to memory. Since most stride values can be represented with fewer than 32 bits, we use only 5 bits to represent it. If that is not enough, then we use the full 32 bit value to represent it. The bit *SV-Type* is used to distinguish the two cases.

The last two fields are used to log the result of the load instruction. Similarly to the stride, most of the load values can be represented using fewer bits than 32 bits. Hence we use 5 bits to represent the load value, and use the additional bit *LV-Type* to distinguish between these two cases as we did with the strides.

```
*************************************************************
Beginning of Checkpoint Interval{
   Initialize header with Thread ID, Process ID,
   Program Counter and Register Values;
   Clear UserMemCache and UserMemHash;
}
*************************************************************
For every LOAD accessing the address Addr{
   If (Addr is found in UserMemCache){
       If(Value for Addr in UserMemCache !=
          Value for Addr in application's memory) {
            Log <Ld-Stride, Ld-Value> in thread's load log;
            Update UserMemCache with (Addr, Value);
       }
   }
   else{
       Update UserMemCache with (Addr, Value);
       If(Miss in UserMemHash OR
            (Value for Addr in UserMemHash !=
             Value for Addr in application's memory)
          ) {
            Log <Ld-Stride, Ld-Value> in thread's load log;
            Update UserMemHash with (Addr, Value);
       }
   }
}
*************************************************************
For every STORE accessing the address Addr:
    Update UserMemCache with (Addr, Value);
*************************************************************
While replacing an Addr in UserMemCache:
    Update UserMemHash with values in replaced block;
*************************************************************
```

Figure 2. Pseudo-code for the per-thread BugNet Logger

4.1.5. Logging of System Call and Interrupt Effects

With the approach described above we are able to capture all the memory side effects from system calls and interrupts. If a system event modifies an already logged memory value, then when that value is loaded again we will see that it differs from the value in *UserMemCache* (or we will have a miss), and the load will be logged again. Since we are able to mirror the copy of the application's memory value in *UserMemCache* structure, we do not have to restart checkpoint intervals on encountering a system call or an interrupt like in the BugNet hardware scheme [17]. But at the same time our mechanism also ensures that our tool is easily portable across different operating systems as it captures the system effects transparently unlike existing deterministic replay debuggers that require redo-logs.

In addition to memory side effects, system calls and interrupts can also modify registers and we need a mechanism to log these changes. To address this, every single time a system call or interrupt occurs we temporarily record all of the register values and the current program counter for the thread.

Upon returning from the system call or interrupt, we then check to see if all of the registers are the same. If any of them are different we log them in a per thread Register Update Log.

There is a register update log per thread. Each entry consists of (a) a memory operation count, (b) register number, and (c) register value. The memory operation count refers to the last memory operation executed in the thread before the system call occurred. Therefore, during replay, when the memory operation count is reached, we will restore the log entries. When this occurs we restore the registers which were modified, along with the PC if it was one of the logged registers. This will effectively skip the execution of the system call, and will deterministically replay execution the same way it occurred during logging.

4.1.6. Logging of Code To Support Self-Modifying Code

In addition to logging the output of the load values, it is also desirable to log information about the code being executed. The advantage of logging the code is that it enables replaying of self-modifying code, which was not supported in our hardware approach [17]. Our logging approach allows us to start executing the program from the beginning of any checkpoint. Therefore, if any code was dynamically generated before that, in a different checkpoint interval, this code would not be available for replay if only the original binaries were used during replay. For this reason in this paper, we also log the code which is executed during a checkpoint interval.

To log the executed code, we use a structure identical to the *UserMemCache* (backed by User-MemHash), but instead of keeping track of the load values, it keeps track of the code executed, similar to an instruction cache. It therefore allows us to keep track of the code as it is seen by the application. Before executing a branch target, we look up the cache structure. If there is a miss, the branch target is logged along with a branch stride value. The branch stride represents the number of branches executed since the last logged code block, similar to the load stride. If there is a cache hit and the contents of the basic blocks mismatch, this means that the code was modified since the last time it was executed and therefore needs to be logged again.

4.1.7. Special Cases

On some processors there are additional instructions that need to be logged, in addition to load values, in order to provide deterministic re-execution. For example, on x86 we also log the output of the Read Time Stamp Counter (RTDSC) instruction, which reads the processor's time stamp counter value into a register. Then during replay, instead of executing this instruction we just set the register to the logged value.

4.2. Replayer Basics

The BugNet logger is used to collect the checkpoint logs for a program and input that has a bug. To provide a debugging tool for users, we also built a deterministic replay debugger tool using Pin [14].

To start debugging an application, the user starts up their program's execution under Pin. For that we still need the original program main image and the dynamic library loader so that pin can start its execution. We also use Pin to reserve the region of memory that was used by the application's dynamic data during logging. We then provide the ability for a user to start replaying the program's execution from any checkpoint. To start execution at a checkpoint, the replayer first initializes the PC and register contents found in the header of the checkpoint log. It then reads the first entry from the code log and initializes a branch count. When the branch count matches the branch stride specified in the log, the code from the log is copied into memory. The branch count is reset and starts counting again from zero.

At the same time the replayer also reads the first entry from the load log and initializes a load count. When the load count matches with the stride value read from the log, the load value is restored to memory, the load count is reset and start counting again from zero. The replayer also maintains a memory count that is used to restore the register state from the Register update log. By repeating the process described, the replayer reads through the checkpoint logs to replay the program.

To start the execution of a checkpoint we use the ExecuteAt API provided by Pin, which allows us to start execution at a specific PC given the current register state. This is useful to restore the checkpoint headers as well as the register updates. The user can then step through the program's re-execution examining the source lines touched and the variables used. If the user is single stepping through the program's re-execution, when they come to the end of a checkpoint, we just start executing the next checkpoint. This will occur when the user comes to an end of a checkpoint interval. In addition, from a users perspective during debugging, the user will just step (skip) over the execution of the system calls and interrupts.

Note that each checkpoint is independent of other checkpoints. Therefore, a user can arbitrarily pick any checkpoint and instantly start the program's re-execution corresponding to that checkpoint. In addition to providing the ability to replay from the beginning of a checkpoint, it is also desirable to step back N instructions at any instant of time. We provide this functionality similarly to how prior work provides the ability to step back N instructions [4].

To summarize, a typical scenario for using BugNet to track down and solve a bug is to first use the BugNet logger to record onto disk the last few percent of the program's execution leading up to the crash or the deviation in program behavior (e.g., wrong answer). The user may then choose to run the program under the BugNet replayer starting at the end of the program's execution to examine the current state when the program ended its execution under the logger. To get to this point the BugNet replayer will quickly *deterministically* replay the last N checkpoints. The user can then see where in the code and the state of memory and stack that was touched during the last few percent of execution. The user can then work their way backward to debug the problem setting backward watchpoints or stepping back N instructions [4], as well as stepping and setting forward watchpoints.

4.3. The Limits of BugNet

We now finish our discussion with the limitations of our BugNet approach.

The focus of BugNet is to assist debugging bugs that do not have complex interactions with operating system routines. Therefore, our approach would not be useful to debug problems in drivers or the operating system, or complex interactions between these and user code.

Even though BugNet cannot replay the system code, it still provides deterministic replay of program execution before and after servicing interrupts and context switches. Hence, the user can examine the values of the parameters passed to the interrupts, and the values loaded and consumed by the user code after servicing the interrupt. This, along with the replay trace, can allow the user to debug some bugs that have interrupt and operating system interactions. Also, we replay all of the operating system shared library code that executes in user space, and the user code along with the OS library code consist of a significant portion of the program's execution. This is sufficient to track down application-level bugs.

5. Support for Multi-threaded programs

The data and code logs collected using our user cache optimization are useful for deterministically replaying any single threaded application. In this section, we describe support for deterministically replaying multi-threaded applications.

Our BugNet checkpoint scheme has an important property: The code and data log for a checkpoint of a thread contains sufficient information to replay that thread independent of other threads. The data logs were collected using the UserMemCache and UserMemHash data structures, which ensure that we have the right values for all the load instructions for each thread, even in the presence of shared memory updates by remote threads. This is because if a remote thread modifies a memory location, then that location's value is going to be inconsistent with the value in UserMemCache/UserMemHash and hence we would log the value updated by the remote thread in the local thread's data log.

Thus, for a multi-threaded program we can still replay each individual thread. However, this ability alone is insufficient to track down race conditions because the programmer needs to know the ordering of the memory operations executed across all the threads. For example, to have the ability to single step through the execution of the multi-threaded program, we need to know the ordering of the memory operations executed across all the threads.

5.1. Logging Multithreaded Shared Memory Dependencies

There are two sub-problems that we need to solve in order to record shared memory dependencies. The first problem is related to detecting these shared memory dependencies during logging. This is especially non-trivial in a software logging tool. The second problem is related to efficiently logging this information in order to reduce the log size.

Previous hardware proposals [29,17] observed that shared memory dependencies can be detected by just observing the coherence messages in a multi-processor system. They used the Netzer transitive reduction algorithm [18] to reduce the log size. In our recent work [16], we observed that we do not have to log all the shared memory dependencies but instead we have to log only the RAW(Read-After-Write) shared memory dependencies. Other forms of dependencies (WAW, WAR) can be inferred using an offline algorithm [16]. Hence, in our software tool, we choose to log only the RAW dependencies, and determine shared WAW and WAR misses offline. Do to space limitations, we only describe how to detect RAW dependencies in software. More details on how to infer WAW and WAR dependencies offline can be found in our other published work [16].

We employ the following algorithm to detect the RAW shared memory dependencies. In the previous section, we described the use of our two data structures, UserMemCache and UserMemHash. The purpose of these two data structures is to keep track of the values seen by the user thread and to detect and log whenever the user data is modified by the system call or the interrupt. We can adapt these structures for multi-threaded programs by having a per-thread UserMemCache data structure and making the UserMemHash structure globally accessible across all the threads. This configuration is similar to the memory hierarchy in a multi-processor system [10]. UserMemCache is similar to a private cache in a processor and UserMemHash is similar to the main memory. Just like in a multi-processor system, we need to keep the UserMemCache of various threads *coherent*. We simulate in the logger the MSI (Modified-Shared-Invalid) coherency protocol [10] to keep the UserMemCache structures coherent. The UserMemHash structure keeps track of the read or write ownership for a block of memory (similar to a directory structure in a multi-processor system). In addition, the UserMemHash structure will have the information about the thread that last wrote to the block (information about the last writer). At any time, there can be multiple threads with read ownership but there can be only one thread with write ownership.

If a load is executed by a thread, and if the private UserMemCache of the thread has the read or the write ownership, then we do not have to record any RAW shared memory dependency. However, if the load address is not found in the private UserMemCache, then UserMemHash needs to be consulted to find the last writer to the block. If the last writer is different from the thread that executed the load, then we detect a RAW dependency and log it. The log is optimized by using a transitive optimization [16].

One of the issues in detecting shared memory dependencies in software is the performance issue due to lock contention to access the shared resource (UserMemHash). This is not a serious issue in our implementation because the UserMemHash structure is accessed only when we encounter a miss in a UserMemCache. There is only a small probability for two threads to encounter a miss in the UserMemCache at the same time.

6. Results and Analysis

In this section we analyze how many instructions need to be logged to capture the root cause of the bug, and the time and space overhead of our software BugNet logger implementation.

6.1. Replay Window Length

The premise of deterministic replay debugging support is that in order to fix a bug, one needs to examine only a window of program execution that immediately precedes the crash. However, it is not clear what the size of this window has to be in order to fix the majority of the bugs. We now quantify the replay window size required for fixing bugs, by matching the execution histories of correct and incorrect program executions corresponding to many open source bugs.

We define the *replay window* length for a bug to be the number of dynamic instructions executed between the point in the buggy program's execution where its output starts to deviate from those of the correct program's execution until the point where the buggy program crashes. In our previous work [17], we provided a very rough estimate of a lower bound for the replay window, which turned out to be too low of a bound to have much meaning. Therefore, one of the goals for this work was to provide a precise measurement of this replay window. One use of this measurement is that it helps us to understand the log space requirements of the BugNet logger and helps us to analyze the efficacy of deterministic replay debugging technique.

In this paper, we would like to present a precise methodology to quantify the replay window length for a bug. In order to determine the replay window length for a bug, we take two binaries corresponding to two versions of the same program. One binary version corresponds to the source code that contains the bug and another version corresponds to the same source code with the bug fixed. We execute these two versions with the same input, which exposes the bug in the buggy program's execution.

Figure 3 shows the buggy behavior of the gzip program with respect to the correct program execution. The x-axis represents the buggy program's execution time. The y-axis represents the magnitude of the difference in the store output values between the correct program's output and buggy program's output, averaged over intervals of 10,000 instructions.

We found that the two executions will follow similar execution path up until a point. After a point in the buggy program's execution, which is really the starting point of the source of the bug, the buggy program's execution deviates from the execution of the correct program. We define the source of the bug to be this deviation point in the buggy program's execution. In order to fix a bug, a developer needs to examine what happened at that deviation point and potentially execution of the program after that point. Therefore, we define our replay window length to be the number of dynamic instructions executed in the buggy program between this deviation point and the end of execution or the point of crash.

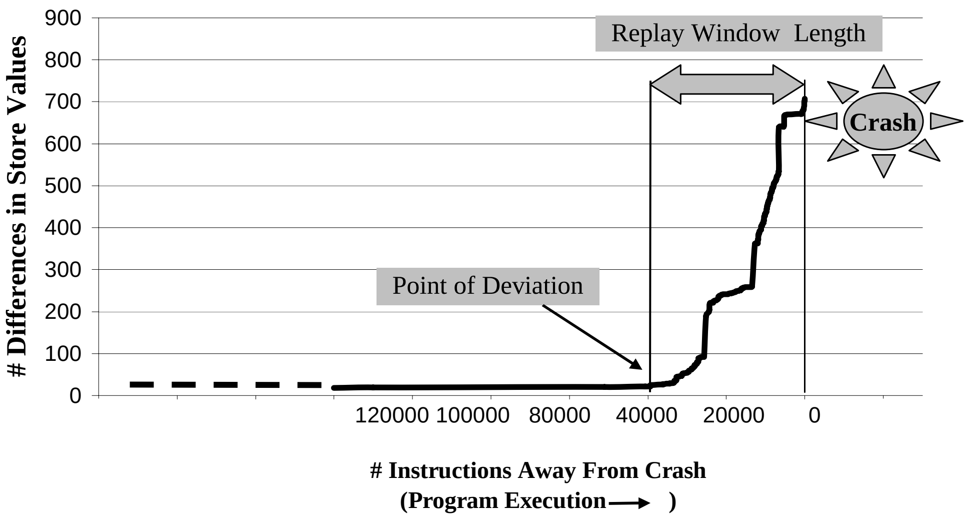

Figure 3. Replay Window Length showing buggy program execution behavior relative to correct execution for gzip

In order to find the deviation point, we collected memory traces from the execution of the buggy program as well as correct program's execution. The memory trace is a trace of store instruction's effective address and the value being stored. We then do a longest subsequence matching between these two traces to determine where they start to deviate focusing on the store values. Our algorithm is similar to the one used by the popular `vimdiff` utility that is used to compare the textual differences between two files. Once we have matched between the two traces as much as possible, we can then determine the deviation point of the buggy program's execution as follows. We compute the number of store output values that differ in the two executions for every interval of 10,000 instructions in the buggy program's execution. This data if plotted will look like the graph shown in Figure 3. We can find the point of deviation by finding the first point of inflection in the graph where the number of differences in store values exceed 30. The number of instructions executed after deviation point till the end of execution of the buggy program will be the replay window length.

Note, our approach for calculating the replay window length is not meant at all to provide a means to debug the program. It is only used to quantify a bound on how much execution may need to be logged. This is important in order to look at the log sizes that will be needed to capture that much execution for our approach.

Figure 4 presents the replay window length required to analyze the bugs in the Siemens benchmark suite [11]. The y-axis shows the number of instructions from the point of deviation until either the program crashed or it terminated with a wrong result. The x-axis shows the results for each of the 100s of bugs examined for this benchmark suite. The result shows that the cause of the majority of bugs (inputs) occurred within the last 1 million instructions of execution. These replay window lengths are for bugs that resulted in the wrong answer when the program finished execution and those that terminated with a crash.

We also studied the bugs found in some popular open source programs. The bugs that we studied are listed in Table 1. The second column in the table gives the details about the location in the source code of the applications that needed to be changed in order to fix the bug. The third column describes the nature of the bug. The set of bugs listed in the Table 1 covers a large variety of bugs. It includes memory corruption bugs like dangling pointer accesses (`ghostscript`), buffer overflow (`gzip`) and null pointer dereferences (`gnuplot`).

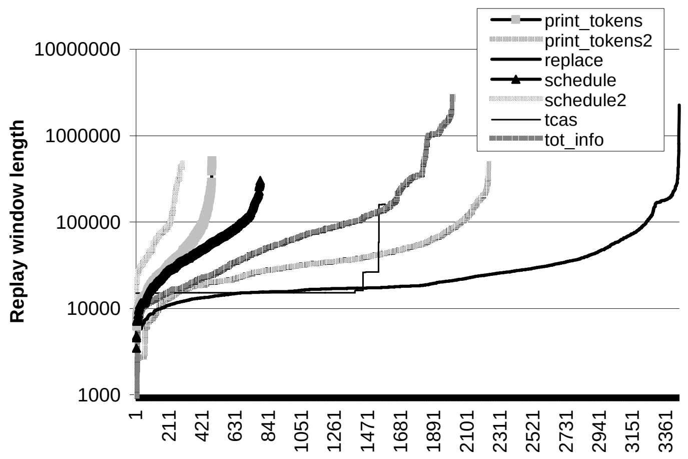

Figure 4. Replay window length required to analyze the bugs in Siemen benchmark suite

Application	Bug Location	Bug Description
bc 1.06	storage.c line 176	Misuse of bounds variable corrupts heap objects
gzip 1.2.4	gzip.c line 1009	1024 byte long input filename overflows global variable
ncompress- 4.2.4	compress42.c line 886	1024 byte long input filename corrupts stack return address
polymorph-0.4.0	polymorph.c lines 193, 200	2048 byte long input filename corrupts stack return address
tar 1.13.25	prepargs.c line 92	Incorrect loop bounds leads to heap object overflow
ghostscript-8.12	ttinterp.c line 5108, ttobjs.c line 279	A dangling pointer results in a memory corruption
gnuplot-3.7.1	pslatex.trm line 189	A buffer overflow corrupts the stack return address
tidy 34132	istack.c at line 31	Null pointer dereference
xv-3.10a	xvbrowse.c line 956, xvdir.c line 1200	A long file name results in a buffer overflow
napster-1.5.2	nap.c line 1391	Dangling pointer corrupts memory when resizing terminal

Table 1. Open source programs with known bugs. The first 5 programs are from the AccMon study [31], and the rest of the programs are from sourceforge.net

Replay window lengths for these open source programs are again determined in the same way as we described earlier. Figure 5 presents the replay window length required to analyze these real bugs. We can note that in the common case the length of the replay window is less than 10 million instructions. The worst case is `ghostscript` for which the replay window length is over 100 million instructions.

6.2. *Dynamic Slicing and Touched Memory*

Dynamic slicing [26,30] is a powerful technique to ease the job of debugging. We implemented dynamic slicing into our replayer, so that the programmer can choose to analyze the execution of only those instructions that produce the value for the instruction that resulted in a crash or an incorrect

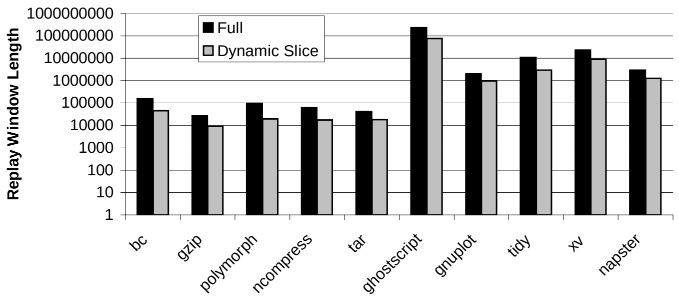

Figure 5. Replay window length required to analyze the bugs in open source programs.

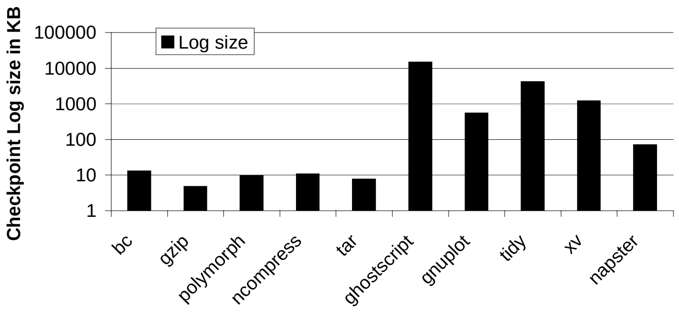

Figure 6. Log size required to capture the replay window.

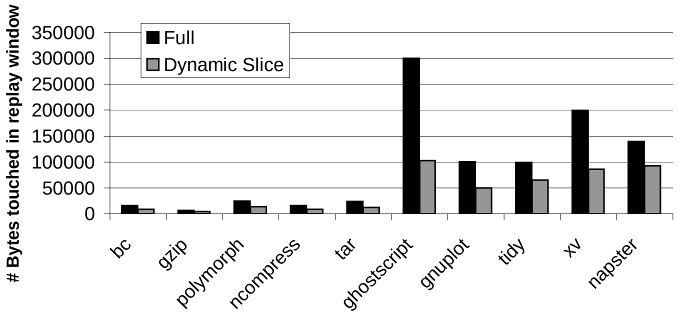

Figure 7. Memory footprint touched within the replay window.

output. The second bar Figure 5 shows the number of dynamic instructions that are on just the dynamic slice. Dynamic slicing results in about one-third reduction in the number of instructions that are required to be examined within the replay window to potentially fix the bug.

We also study the number of memory locations touched by the programs within the replay window. Results for this study are shown in Figure 7. This shows the number of unique memory bytes touched in the replay windows for both the full replay window and just the dynamic slice. Using the dynamic slice can reduce the amount of memory locations that might need to be examined by about a half.

6.3. *BugNet Log Size and Performance*

In order to analyze the space and time overhead of our software BugNet logger we use the SPEC [24] benchmark suite, which is widely used for performance evaluation. Figure 8 shows the checkpoint log size requirements of the software BugNet logger and the Figure 9 shows the performance overhead involved in collecting these logs. These results are for a checkpoint interval of size 100 million instructions - that is, the data structures UserMemCache and UserMemHash are cleared after executing a checkpoint interval comprising of 100 million instructions.

Figure 8 shows that on average we require less than 10MB of BugNet checkpoint logs to have the ability to replay 100 million instructions for SPEC programs. We used a 32 KB UserMemCache structure. Structures larger than this adversely impacted the overhead of the logger. In addition, we used a UserMemHash structure, which helps reduce the log sizes, but we disabled the UserMemHash structure whenever we find that the performance of the logger overhead was too high.

We now show the amount of log size (without compression) that needs to be collected in order to analyze the open source bugs in Figure 6. For the largest replay window needed we required about 10 MB of log size, which is enough to capture and replay 100 million instructions in the case of `ghostscript`.

Figure 9 shows the execution time overhead of the software BugNet logger normalized to the execution time of the program when it is executed natively on the system. The program `vortex` is our worst case that incurs about 158x slowdown, but on average the performance overhead of the logger is 86x.

Figure 10 shows the execution time overhead for multi-threaded applications selected from the SPLASH benchmark suite [28]. These programs run for only about 10 seconds natively. Due to the short execution time, the baseline instrumentation overhead itself (without any BugNet logging support) dominates for these applications as shown in Figure 10. On average, we experience about 250x slowdown, of which about 150x is due to the baseline instrumentation overhead.

7. Conclusion

Program developers can benefit greatly from a debugger that would allow them to deterministically reproduce the execution of programs as many times as required. In this paper we presented our software BugNet logger and debugger that allows programmers to deterministically replay application code and shared libraries in the presence of interrupts, system calls and synchronization operations.

A key mechanism that we used is the load-value based checkpoint scheme. By keeping a shadow copy (the UserMemCache) of a thread's already logged execution state, we can automatically detect all the system and shared memory side effects. This allows us to capture all the system effects in a log without the knowledge of the underlying operating system. The log can be used to deterministically replay a thread's execution in any environment, which solves the issues involved with capturing and reproducing bugs.

Acknowledgments

We would like to thank the anonymous reviewers for providing useful comments on this paper. This work was funded in part by grants from Intel and Microsoft.

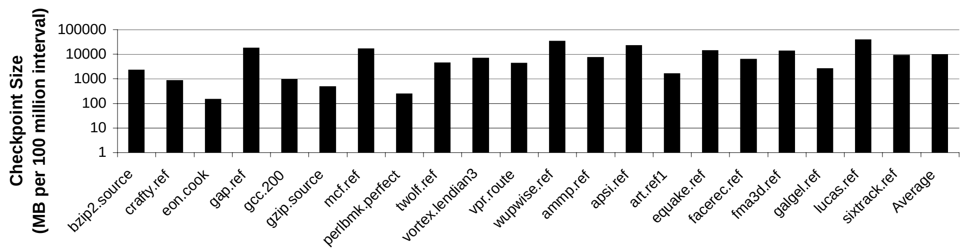

Figure 8. Logsize overhead for SPEC, shown in terms of MB per 100 million checkpoint interval

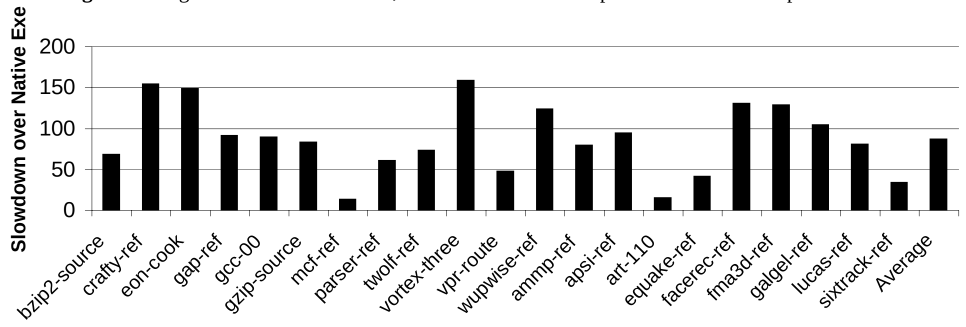

Figure 9. Execution time overhead for SPEC

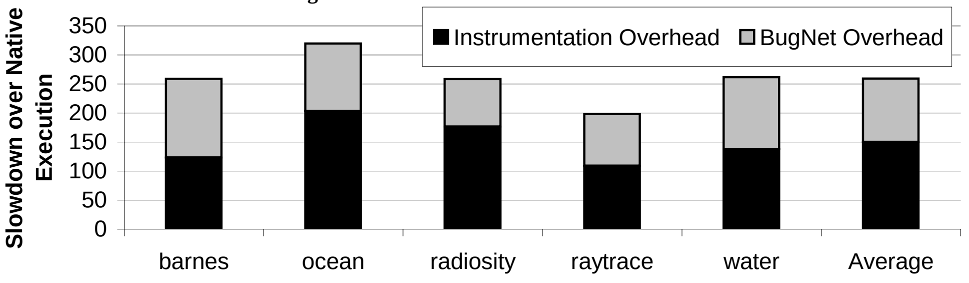

Figure 10. Execution time overhead for SPLASH multi-threaded programs

References

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. An execution backtracking approach to debugging. In *IEEE Software*, May 1991.

[2] D. F. Bacon and S. C. Goldstein. Hardware assisted replay of multiprocessor programs. In *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 194–206. ACM Press, 1991.

[3] S. Bhansali, W. Chen, S. D. Jong, A. Edwards, and M. Drinic. Framework for instruction-level tracing and analysis of programs. In *Second International Conference on Virtual Execution Environments*, June 2006.

[4] B. Boothe. Efficient algorithms for bidirectional debugging. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 299–310, New York, NY, USA, 2000. ACM Press.

[5] S. Chen, W. K. Fuchs, and J. Chung. Reversible debugging using program instrumentation. *IEEE Trans. Softw. Eng.*, 27(8):715–727, 2001.

[6] J. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools, pp. 48-59, Welches, Oregon*, 1998.

[7] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, 2002.

[8] S. I. Feldman and C. B. Brown. Igor: a system for program debugging via reversible execution. In *PADD '88:*

Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging, pages 112–123, New York, NY, USA, 1988. ACM Press.

[9] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, 1992.

[10] J. Hennessy and D. Patterson. *Computer Architecture a Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1996.

[11] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[12] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX 2005 Annual Technical Conference*, 2005.

[13] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transaction on Computers*, 36(4):471–482, 1987.

[14] 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 *Programming Language Design and Implementation*, Chicago, IL, June 2005.

[15] E. Marcus and H. Stern. *Blueprints for high availability*. John Willey and Sons, 2000.

[16] S. Narayanasamy, C. Pereira, and B. Calder. Using global markers and an offline analysis to capture shared memory dependencies. In *Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct 2006.

[17] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *32th Annual International Symposium on Computer Architecture(ISCA-32)*, Madison, WI, June 2005.

[18] R. H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proceedings of the 1993 ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 1–11. ACM Press, 1993.

[19] RTI. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. NIST, Research Triangle Park, NC, May 2002.

[20] M. Russinovich and B. Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 258–266, New York, NY, USA, 1996. ACM Press.

[21] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[22] E. Schatz and B. G. Ryder. Directed tracing to detect race conditions. In *ICPP (2)*, pages 247–250, 1992.

[23] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the USENIX'05 Annual Technical Conference*, April 2005.

[24] SPEC. Standard performance evaluation corporation - http://www.spec.org.

[25] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conference, General Track*, pages 29–44, 2004.

[26] F. Tip. Generic techniques for source-level debugging and dynamic program slicing. In *TAPSOFT '95: Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 516–530, London, UK, 1995. Springer-Verlag.

[27] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[28] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *22nd Annual International Symposium on Computer Architecture*, pages 24–36, 1995.

[29] M. Xu, R. Bodik, and M. Hill. A flight data recorder for enabling full-system multiprocessor deterministic replay. In *30th Annual International Symposium on Computer Architecture*, San Diego, CA, 2003.

[30] X. Zhang, H. He, N. Gupta, and R. Gupta. Experimental evaluation of using dynamic slices for fault location. In *Sixth International Symposium on Automated and Analysis-Driven Debugging*, 2005.

[31] P. Zhou, W. Liu, F. Long, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. Accmon: Automatically detecting memory-related bugs via program counter-based invariants. In *37th International Symposium on Microarchitecture (MICRO)*, Nov 2004.