BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging

Satish Narayanasamy Gilles Pokam Brad Calder

Department of Computer Science and Engineering
University of California, San Diego
{satish,gpokam,calder}@cs.ucsd.edu

Abstract

Significant time is spent by companies trying to reproduce and fix the bugs that occur for released code. To assist developers, we propose the BugNet architecture to continuously record information on production runs. The information collected before the crash of a program can be used by the developers working in their execution environment to deterministically replay the last several million instructions executed before the crash.

BugNet is based on the insight that recording the register file contents at any point in time, and then recording the load values that occur after that point can enable deterministic replaying of a program's execution. BugNet focuses on being able to replay the application's execution and the libraries it uses, but not the operating system. But our approach provides the ability to replay an application's execution across context switches and interrupts. Hence, BugNet obviates the need for tracking program I/O, interrupts and DMA transfers, which would have otherwise required more complex hardware support. In addition, BugNet does not require a final core dump of the system state for replaying, which significantly reduces the amount of data that must be sent back to the developer.

1 Introduction

The last few decades witnessed exponential growth in processor performance. Sophisticated software systems were built to leverage the performance growth, but unfortunately at the cost of software reliability. The Tandem OS had 4 million lines of code in 1986, while Windows XP had 40-50 million lines of code in 2003 [4]. Unfortunately, software engineering and verification techniques are complex, and a software system even after undergoing rigorous Quality Assurance is bound to have bugs. In addition, with the explosion of the Internet, it has become trivial to distribute software patches, and this has encouraged vendors to have more rapid release schedules, which results in released software with more bugs.

Tracking down and fixing bugs in released software can be a nightmare, costing a significant amount of time and money. The main difficulty lies in being able to reproduce the bug at the developer site. Most of the current debugging systems rely on a core dump [8, 7], which represents the final state of the system before the crash. Unfortunately, this solution has been woefully inadequate as it is difficult to determine the source of error that was actually responsible for the program to end up with the state represented by the core dump.

A good solution to this problem will be to provide architecture support to capture enough information during a production run that can be used by the developer to deterministically replay the program's execution like it happened moments before the program crashed. *Deterministic Replay Debugging* (DRD) is the ability to replay exactly the same sequence of instructions that led up to the bug. We believe DRD to be an effective way to isolate the source of the problem and fix it. The Flight Data Recorder (FDR) [24] system is one of the first proposals toward an architecture to enable DRD.

FDR builds on top SafetyNet [21]. SafetyNet is a checkpointing scheme used to provide reliability in shared memory multiprocessor systems. FDR employs additional hardware to track data races, program I/O, interrupts and DMA accesses which are all required for deterministic replay of the full system from the beginning of the checkpoint. Using the information recorded in FDR, it is possible to replay the entire system execution, which includes interrupt handler and system call routines, in addition to user code.

In this paper we present an architecture focused on continuously tracing program execution called BugNet. BugNet focuses on deterministically replaying the instructions executed in user code and shared libraries. A significant number of application level bugs can be replayed using this approach, which makes it an attractive software development tool. Since BugNet focuses only on application level bugs, it cannot replay the full systems execution as in FDR. Even so, BugNet allows replaying across interrupts and system calls, and the log sizes generated and amount of hardware used is significantly less than FDR, since the focus is on application level bugs.

2 Related Work

Microsoft's Dr. Watson tool [8] and Mozilla's Talkback [7] are examples of current solutions to gather and analyze the reason for a program crash. All these tools collect information that represent the final snapshot of execution state when the program crashed. While these crash reports have some utility, it is highly desirable to have the ability to exactly replay the sequence of instructions executed before the crash. An example of such a replayer is from Ronsse and De Bosschere [19]. They built a software debugger that logs all the information that would enable one to go back in time and repeatedly re-execute the program

0-7695-2270-X/05/$20.00 (C) 2005 IEEE

through the same sequence of instructions.

2.1 Checkpoint Schemes

To enable replaying, much of the prior work uses some form of checkpointing. There is a body of prior work on various checkpointing schemes for designing Checkpoint and Restart (CPR) systems to improve reliability [10]. On encountering a fault (transient error in hardware or a software error) the system can rollback to the previous checkpoint and continue execution from thereon. The purpose of checkpointing schemes in CPR systems is to provide a mechanism to retrieve a consistent full system state from where execution can be continued without error. They usually involve mechanisms to retrieve the full system state at the beginning of a checkpoint. CPR could be done either with OS support [9] or hardware support as in SafetyNet [21] and Revive [16].

FDR [24] adopts the SafetyNet [21] checkpoint mechanism to retrieve a consistent full system state corresponding to a prior instance in time. Additionally, it records all the inputs coming into the system (I/O, interrupts, DMA transfers) to enable replaying. With this recorded information, starting from the retrieved full system state, the original program execution can be replayed.

The checkpointing scheme we propose does not retrieve a consistent full system state. Instead it concentrates on just tracking the input to the program read through load instructions. We show that this information, along with a record of the register state at the beginning of a checkpoint, is sufficient to replay the instructions executed. Since our checkpoint scheme does not recreate a consistent full system state it is not suitable for using in Checkpoint and Restart systems, but is useful for reproducing application level bugs.

2.2 Deterministic Replayers for Debugging Multithreaded Applications

In the past, there have been studies that presented deterministic replayers where the goal was to provide support to debug multithreaded applications. InstantReplay [11] is a software based deterministic replayer that records the memory access order, and a hardware-assisted version was presented by Bacon and Goldstein [1]. The amount of information that needs to be logged to record memory access ordering can be reduced by applying transitive properties [14]. If one assumes that the multithreaded application is running on a single processor, then one can limit the amount of recording to just the scheduler decisions [5]. Another alternative is to just record the control flow of the threads [23].

There are also proposals to record the necessary information to detect data races. RecPlay [18] logs synchronization races and the races are detected through offline analysis. Eraser [20] takes a slightly different approach in that it tries to find the errors arising due to incorrect usage of locks for shared variables. ReEnact provides an approach for rolling back and replaying the execution using thread level speculation support [15]. In ReEnact, upon detection of data races, it can rollback to a previously logged checkpoint and replay the execution. Flashback [22] provides lightweight OS support for fine grained rollback and replaying the program execution.

As opposed to all of the these techniques, only FDR provides a complete history of what happened before the program crash that enables one to deterministically replay the program execution [24]. This is needed in order to communicate back to the developer a trace to faithfully reproduce the problem.

2.3 Architecture Support for Debugging

Recently there has been a great interest in providing support to enable better software reliability and correctness. iWatcher [27] provides sophisticated watchpoints to debug applications. It associates tags with memory locations, and when these locations are accessed, a specific function is executed to perform monitoring. SafeMem [17] and AccMon [26] are other recent proposals that provide architectural support to catch memory violations dynamically during program execution. Architecture support for efficiently implementing breakpoints and watchpoints in an offline debugger is also discussed by Corliss et. al. [6].

3 Flight Data Recorder

The goal of FDR [24] is to provide architectural support for collecting enough information to enable one to deterministically replay the last one second of full system execution before the crash. FDR strives to replay the full system by having support to replay interrupts and system call routines. In contrast, BugNet limits its focus on replaying only execution in user code and shared libraries, but it can still deterministically replay the application across interrupts.

FDR continuously records three kinds of information:

- Checkpoint - FDR uses the SafetyNet checkpoint mechanism [21] to retrieve a consistent system state to start the replaying of execution.
- Interrupts and External Inputs - In order to perform deterministic replay, FDR records all the interrupts, memory mapped I/O and DMA transfers.
- Memory Races - To support debugging data races in multithreaded applications, shared memory access ordering is recorded using an additional Memory Race Buffer (MRB).

The combined sizes of logs needed for replay in FDR is about 34 MB [24]. The hardware complexity in FDR to collect all the required information to enable full system replay is about 1.3 MB of on-chip hardware and 34 MB of main memory space. In addition, the final snapshot of the entire physical memory image, to be communicated to the developer, can be on the order of 1 GB, depending on the memory footprint of the application and the main memory sizes used. Since BugNet focuses on just capturing application level bugs, the amount of information that needs to be recorded for replay can be significantly reduced. In addition, smaller trace sizes will encourage even those users with network bandwidth constraints to communicate the trace back to the developer.

4 BugNet Architecture

BugNet provides architecture support to record enough information to deterministically replay a window of instructions preceding a program's crash. We first give an overview of the architecture, and then discuss each component in detail.

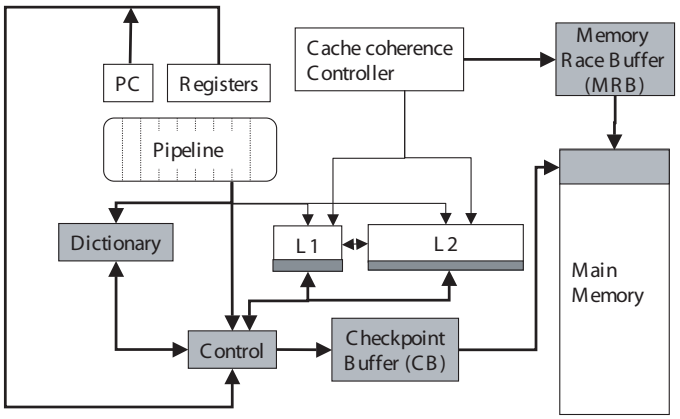

Figure 1: BugNet Architecture.

4.1 BugNet Architecture Overview

The goal of BugNet is to record the information that needs to be communicated back to a developer to enable deterministic replay debugging. Unlike FDR [24], we limit our focus on replaying just the user code and shared libraries to enable the debugging of application level bugs. BugNet supports deterministic replay across interrupts and system calls, but the developer will not be able to analyze what goes on during the execution of these interrupts and systems call routines.

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. Therefore, a *checkpoint interval* represents a window of committed instructions that is captured by the checkpoint being logged. During tracing, a maximum size (number of committed instructions) is specified for the checkpoint interval. When this limit is reached, a new checkpoint interval is initiated, but a checkpoint can be prematurely terminated on encountering an interrupt or a context switch. During 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.

BugNet is built on the observation that a program's execution is essentially driven by the values read when executing load instructions. Hence, in order to replay a checkpoint interval, one needs to record just the initial register state and then record the values of load instructions executed in that interval. During a checkpoint interval, the memory values could have been modified by interrupts, especially I/O interrupts and DMA transfers. In addition, in the case of shared memory multithreaded programs, the shared memory can be modified by other threads during a checkpoint interval. However, by logging the load values, we ensure that we have recorded information required for deterministic replay including any memory values that were updated by the interrupt handlers or by other threads in a shared memory processor.

Figure 1 represents the major components in the BugNet architecture. Components shaded in gray are the new additions to the baseline architecture. BugNet operates by creating checkpoints at the beginning of checkpoint intervals. At the start of a checkpoint interval, a snapshot of the architectural state is recorded in the *Checkpoint Buffer* (CB). The recorded architectural state includes the program counter and register values. After initialization, whenever a load instruction is executed a new log entry is created to record the load value and this is also stored into the CB.

Recording the result value of every load instruction would clearly be expensive. But we note that a load value needs to be recorded only if it is the first access to the memory location in the checkpoint interval. The result value of the other loads can be trivially generated during replay. Therefore, we associate a bit with every word in the cache, to identify these temporal hits, which are not recorded. When applying this optimization, special care needs to be taken to handle interrupts and shared memory accesses by remote threads. If a memory address is modified by these external entities, we make sure that the future load reference to that address is logged. In order to further optimize the trace size, we employ a dictionary based compressor shown in the Figure 1, which exploits the frequent value locality in load values [25].

The log that contains the load values and the initial architectural state information for a checkpoint interval is referred to as the *First-Load Log (FLL)*. The information recorded in the FLL for a checkpoint interval is sufficient to replay that interval. This allows us to re-execute the instructions with exactly the same input and output register and memory values as in the original execution. This is true even in the case of multithreaded programs, because the FLL for a thread contains the necessary information to replay each thread independent of the other threads. In order to debug data races between these threads, we need to record synchronization information to reproduce the ordering of memory operations across the threads. We use the *Memory Race Buffer* (MRB) to record synchronization information building on the approach used in FDR [24], which are kept in *Memory Race Logs (MRL)*.

The CB and MRB are FIFO queues that are memory backed to allow the collection of larger logs than can be stored in these dedicated hardware buffers. The operating system provides support for managing the memory space that needs to be allocated for BugNet's use. The goal is to continuously log program execution into memory overwriting older logs in memory, and to only dump the logs to disk when a bug is found. Therefore, the memory can contain logs corresponding to multiple consecutive checkpoints and logs from many different threads. When the allocated memory space fills up, the logs corresponding to the oldest checkpoint for a thread are discarded.

Enough FLLs are kept in memory for the threads being traced to at least replay tens of millions of instructions of execution for each thread. A developer is interested in replaying enough instructions to determine the cause of a bug. This number of instructions should be less than the replay window, captured by BugNet. The *replay window* is the number of instructions that can be replayed for a thread given a sequence of FLLs and MRLs stored in memory.

When the operating system detects that a program has encountered a fault, before terminating the application it first records the instruction count of the current FLL and the program counter of the faulting instruction in the current FLL and then

stores all the logs collected for that application to a persistent storage device. The logs are then sent back to the developer for debugging.

4.2 Checkpoint Scheme

For checkpointing, the program's execution is divided into multiple checkpoint intervals, where the interval length is specified in terms of the number of instructions executed. At the end of a program interval, the current checkpoint would be terminated and a new one will be created. In addition, interrupts and system calls can also terminate a checkpoint interval, which we will describe later. Finally, a fault in execution will terminate the checkpoint interval and initiate the collection of logs to be sent back to the developer for debugging.

A new checkpoint is recorded by creating a new FLL deliminator in the Checkpoint Buffer (CB) and initializing the checkpoint interval's FLL with the following header information:

- **Process ID and Thread ID** - are required to associate the FLL 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 beginning to replay the program execution using the recorded load values.
- **Checkpoint Interval Identifier (C-ID)** - is used to identify the checkpoint interval (and MRL) corresponding to this FLL.
- **Timestamp** - is the system clock timer when the checkpoint was created. This is useful for ordering the FLLs collected for a thread according to their time of creation.

When a checkpoint interval is created, a C-ID is generated to provide a unique identifier for the interval. The C-ID is incremented whenever a new checkpoint interval is created. The number of bits used for this counter and the C-ID is dependent on the maximum number of checkpoints that can reside in memory at any instant of time. The counter is set to zero when it overflows.

The FLL after getting initialized with the above information will be appended with the output values of the load instructions executed during the checkpoint interval.

4.3 Tracking Load Values

Within a checkpoint interval, a load accessing a memory location needs to be logged only if it is the first access to that memory location. The values of other loads can be re-generated during replay. Logging just the "first-loads" to a memory location will significantly reduce the number of load values that need to be recorded. In order to do this optimization, we associate a first-load bit with every word in the L1 and L2 caches. At the start of a checkpoint interval all these bits will be cleared. When a load accesses a word for which the bit is not set, then the load value will be logged in the Checkpoint Buffer, and the bit will be turned on. If the bit is set for a word in the cache, then it implies that the value of that word has already been logged and hence future load accesses to it need not be logged.

This approach is adapted from FDR [24]. FDR's goal was to track during a checkpoint interval the first store to a particular location and to log the value that is overwritten. They focus on stores, since they use the store values to repair the final core image to retrieve memory state at the start of a checkpoint interval. For BugNet, if the first access to a particular memory location is a store then we would set the bit and not log the value of the store. The future load accesses to this memory location would not be logged as well, as the bit would be set. The store values are not logged, and this mechanism works because the stores will be generated by the execution of instructions during replay.

When a cache block is replaced from the L2 cache, all the first-load bits associated with the words in that cache block will be cleared. Therefore, logged values for addresses (blocks) evicted from the L2 will be relogged when the block is brought back in and those same addresses are accessed again. The first-load bits in the L2 cache are used to initialize the first-load bits in the L1 cache when bringing in a block to the L1 from the L2. When an L1 block is evicted, its first-load bits are stored into the first-load bits of the L2 cache.

The above first-load optimization will be effective for long checkpoint intervals. This is because the greater the number of loads/stores 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.

During replay, we need to determine whether the value for a load instruction is recorded in the log or not. If it is recorded, then the load executed during replay needs to get its value from the FLL. If it was not recorded then it is certainly not the first access to the memory location that it is accessing. By simulating memory state during replay, the value can be obtained by reading from the simulated memory state. To determine when to consume a load value during replay, as part of each log entry we have a field to specify the number of load instructions that were skipped since the last load instruction was logged. The following is the format of each log entry in the checkpoint to record the information for the load instruction:

```
(LC-Type, Reduced/Full L-Count,
 LV-Type, Encoded/Full Load-Value )
```

The second field, *Reduced/Full L-Count*, represents the number of load instructions skipped (not logged) between the current load instruction being logged and the last load instruction logged. To record the full L-Count value, one would require `log(checkpoint interval length)` bits, since the L-Count cannot be greater than the maximum checkpoint interval size used. We found that the majority of L-Count values can be represented using just 5 bits. Hence, we record L-Count using 5 bits whenever its value is less than 32. If the L-Count value exceeds 32, then we resort to recording the full L-Count value. The logs that contain the full L-Count values are distinguished from the logs that contain the 5-bit L-Count values by using one additional bit, the *LC-Type*.

The fourth field, *Encoded/Full Load-Value*, is used to record the load value. Again, we try to avoid recording the full 32-bit load value. To achieve this, we use a 64-entry dictionary that

captures the most frequently occurring load values. If the load value is found in the dictionary, then we use 6 bits to represent the position of the value in the dictionary. If the load value is not found, then the full 32-bit value is recorded. *LV-Type* is the bit that is used to distinguish between the two cases.

To track load instructions we just record their output values in the log. Neither the effective address nor the address of the PC of the load instruction is logged, since they can be produced during replay of the thread's execution. In Section 5 we describe the replaying mechanism in detail.

4.3.1 Dictionary-based Compressor

In BugNet, the load values are compressed using a dictionary based compression scheme. It has been shown in [25] that the load values exhibit frequent value locality. That is, over 50% of all the load values can be captured using a small number of frequently occurring values. In addition, value predictors have been shown to provide impressive compression ratios [3].

In our approach, a 64-entry fully associative table, called the *Dictionary Table*, is used to capture these frequently occurring load values. The dictionary table is emptied at the beginning of the checkpoint interval and is updated with the execution of each load instruction. Before logging a load value into the FLL, the value is looked up in the dictionary table. If there is a hit, instead of storing the full 32-bit value, we will store a 6-bit encoding. The 6-bit encoding corresponds to the rank of the value in the dictionary table. In our design, the rank corresponds to the index into the dictionary table used to find the matching value.

In a checkpoint interval, the dictionary table will be continuously updated as load instructions are executed. As a result, the position of a value in the dictionary table can keep changing during an interval. Therefore, the encoding that we use to compress a value can change over the course of a FLL. This is valid since we simulate the dictionary state during replay. During replay we know the initial dictionary state (which is the empty state) at the start of a checkpoint interval, and all the subsequent executed load instructions update the table. Therefore, at any instant of time, while executing a load instruction during the replay, the state of the dictionary table will be the same as its state during the original execution.

For *every* load that gets executed within the interval, the dictionary table will be updated as follows. Each entry in the table has a 3-bit saturating counter to keep track of the frequency of the value stored in the entry. When a load value is found in an entry in the dictionary table, the 3-bit saturating counter corresponding to that entry is incremented until it saturates. If the updated counter value is greater than or equal to the counter value of the previous entry in the table, then the two values will swap their positions (rankings) in the table. This ensures that very frequently occurring values will percolate to the top of the table. When a load value is not found in the dictionary table, then it is inserted into the entry with the smallest counter value. If there are multiple candidates, then the entry occupying the lowest position in the table is chosen for replacement.

4.4 Handling Interrupts and Context Switches

Interrupts can be either asynchronous or synchronous. Asynchronous interrupts are caused by sources external to the executing application code, like I/O and timer interrupts. On the other hand, synchronous interrupts (also commonly referred to as traps) are triggered while executing program instructions. Reasons for traps include arithmetic overflow exceptions, invoking a system call or an event like a page fault.

Since our goal is to replay and debug only the application code, we do not record what goes on during any interrupts. So we do not record the output of load instructions executed as part of the interrupt handler and operating system routines servicing the interrupts. Nevertheless, we need to track how the interrupt affects the execution of the application. Interrupts are likely to modify the memory state (e.g. I/O interrupt) and they can even change the architectural state of the program's execution by modifying the program counter or registers.

A straight forward solution is to solve this problem by prematurely terminating the current checkpoint interval on encountering an interrupt and create a new one when the control returns to the application code. If we create a new checkpoint after servicing the interrupt, we are guaranteed to have the right program counter value, as it will be initialized in the header of the new FLL. Also, the bits used to track the first-loads would have been reset, thus ensuring that the necessary load values are logged to replay the instructions that were executed after the interrupt. The architecture that we model while discussing our results in Section 6 assumes this approach.

A more aggressive solution would be to allow the first-load bits to be tracked across the checkpoints and interrupts. This would help in reducing the number of load instructions logged, when restarting a new checkpoint after the interrupt. The solution needs to make sure that the first-load bits are correctly invalidated when the memory state is updated during the interrupt or context switch. Examining this approach in more detail is left for future research.

4.5 Handling External Input

The mechanism described in the previous section to handle interrupts is adequate to handle I/O interrupts as well. A memory mapped I/O mechanism works by mapping the address space of a device to the program's virtual address space. The values are read from the device through load instructions using the virtual address corresponding to the program's address space.

The OS can initiate a DMA transfer to service an I/O system call. In such cases, the control will return to the application code but the DMA transfer can proceed in parallel. Like in FDR [24], we assume that a DMA write would use a directory based cache coherence protocol and invalidate the cache block in the processor executing the application. This would ensure that the bits used for the first-load optimization are reset and hence the values in the modified memory location would be recorded later when they get referenced by an application load.

Our scheme of recording only the first-load values avoids logging the data copied into the process's address space until it is referenced in the application. Even though a large amount of data can get copied into the process's address space, not all of it will necessarily be used by the program execution preceding the crash. Our scheme ensures that we log just the necessary values that are in fact consumed by those instructions that need to be replayed later.

4.6 Support for Multithreaded Applications

We assume a shared memory multiprocessor system to execute multithreaded programs. In shared memory multithreaded applications, remote threads executing on other processors can modify the shared data within a checkpoint interval. This problem is the same as the one that we discussed regarding DMA transfers. When a shared memory location is modified by a remote thread, the corresponding cache block will be invalidated before the update. This would reset all the bits used to track first-loads to that cache block. As a result, future load references to the same cache block would result in recording the value written by the remote thread in the FLL.

The FLL corresponding to a checkpoint interval is sufficient to replay the instructions executed in that interval. This is true even in the case of multithreaded applications. Any thread can be replayed independent of the other threads as we would have recorded all the input values required for executing that thread. However, in order to assist debugging data races one would require an ability to infer the order of instructions executed across the threads. To record synchronization information we adapt the mechanism proposed in FDR [24], and record them in a separate Memory Race Buffer (MRB).

4.6.1 Memory Model

As in FDR, we assume a shared memory multiprocessor with a sequential consistency memory model using a directory based cache coherence protocol. In a sequential consistency memory model, all the memory accesses appear to occur one at a time. Thus, the execution of the program can be represented by interleaving the instructions from different threads into a sequential order. In order to get this valid sequential order during replay, we adapt the mechanism used by FDR [24] to record the order of memory operations across the threads.

4.6.2 Asynchronous Checkpointing in Multithreaded programs

FDR's [24] checkpoint mechanism uses barrier synchronization to support shared memory multithreaded applications. The mechanism ensures that the checkpoint intervals across all the threads start at the same instant of time. This approach is not desirable in our BugNet architecture because of its overhead in terms of performance, especially when we want to create checkpoints with a smaller interval length. Moreover, we want to have the flexibility of terminating a checkpoint independent of other threads, like when interrupt events are encountered. Hence, we allow the threads to create and terminate checkpoints intervals independent of the other threads. As a result, the checkpoint intervals across different threads may not start at the same time. To support asynchronous checkpoints across threads, we record checkpoint identifiers as part of every memory race log entry, as described in the next section.

4.6.3 Memory Race Log

Whenever a new checkpoint is created, in addition to creating a new FLL as explained before, we also create a new Memory Race Log (MRL) stored in the memory race buffer. Every thread records synchronization information in its local MRL. For a given thread, the MRLs are kept in synchronization with the FLLs. When a new checkpoint interval is created, in addition to creating a new FLL, a new MRL is also created and is initialized with the following header information:

- **Process ID and Thread ID (T-ID)** - is used to associate this memory race log with the thread that it corresponds to.
- **Checkpoint Interval Identifier (C-ID)** - is used to identify the checkpoint interval (and FLL) corresponding to this memory race log.
- **Timestamp** - is the system clock timer when the checkpoint was created. The Timestamp is used to order the MRLs during replay across the different threads.

The goal of the memory race log is to track shared memory ordering among threads. FDR's proposal for creating an MRL is to piggy-back the coherence reply messages (write invalidation acknowledgment while executing writes and write update reply while executing load) from the remote thread with its execution state. An entry in the MRL would be created whenever there is a coherence reply message for a shared memory access. No MRL entry would be logged while executing a load or store to a memory location that is in non-shared or exclusive state, since no coherence reply would be received for those memory operations.

For BugNet, an MRL is initialized with the above information at the beginning of a checkpoint interval. After that, the log is appended with the following information when a coherence reply message is received:

```
(local.IC, remote.TID, remote.CID, remote.IC)
```

The purpose of each record is to synchronize the execution of the remote thread, which is sending the coherence reply message, with the execution of the local thread by recording both of their instruction counts. Having this information allows us to retrieve the ordering of memory operations across all the threads. The remote thread sends its execution state, as part of its coherence reply, to the local thread that executed the memory operation.

The $local.IC$ represents the number of instructions executed from the beginning of the current checkpoint interval till the current memory operation in the local thread. Hence the size of $local.IC$ needs to be only as large as `log(checkpoint interval length)`. The other three fields are used to represent the state of a remote thread. The $remote.TID$ identifies the thread ID. The size of this field is as large as `log(max live threads)`. The $remote.CID$ represents the checkpoint interval identifier corresponding to the checkpoint interval that is currently active in the remote thread. The number of bits required for the checkpoint identifier is dependent on the number of checkpoints that can simultaneously reside in memory. The $remote.IC$ represents the instruction count of the remote thread when the coherence reply is sent. A detailed description of how to use the FLLs with the MRLs to order the replaying of multiple threads will be discussed later in Section 5.2.

FDR optimizes the size of memory race logs by providing architecture support to implement Netzer's algorithm [14], which we also assume. See FDR [24] for the details on this memory race log optimization.

4.7 Memory Backing

The First-Load Logs stored in the Checkpoint Buffer and the Memory Race Logs stored in the Memory Race Buffer (MRB) are memory backed at two different locations in memory. The amount of memory space devoted for this purpose can be managed by the user and/or the operating system to ensure that the performance impact is within tolerable limits. The amount of memory space and disk space devoted will determine the number of instructions that can be replayed.

The contents in the buffer are lazily written back to memory whenever the memory bus is idle. Since we can compress the log entries as they are generated, the contents in the buffer can be lazily written back to memory at any point. This memory backed solution can potentially impact the memory bandwidth requirements due to extra traffic to main memory. When the processor is accessing main memory on encountering a cache miss, it will most probably be stalled waiting for data to be obtained from the main memory. Thus, the rate at which loads are logged in the FLL will be reduced. We found when simulating the SPEC benchmarks that there is sufficient bandwidth to write the logs back to memory when the memory bus is idle, and the on-chip buffers need to be only large enough to hold bursts in the logging.

4.8 On Detecting a Fault

The operating system will know when the program executes an instruction that causes the thread to be terminated. An arithmetic exception due to division by zero or a memory operation accessing an invalid address are some examples that can trigger the program to crash. Once the operating system detects that the program has executed a faulting instruction, it records in the FLL the current instruction count in the checkpoint interval and the program counter of the faulty instruction. This is used to determine when to stop replaying and to correctly identify the faulty instruction.

In addition, the OS collects the FLLs and MRLs corresponding to the application from the main memory and hardware buffers. It scans through the headers of all the logs and uses the process identifier in the header to identify the logs that correspond to the application. The collected logs are then stored to persistent storage and sent to the developer for debugging.

5 Replayer

We implemented a prototype of a replayer as a proof of concept for BugNet using Virtutech Simics [13]. In this section we will describe how the BugNet replayer works and our experiences in building the replayer system using Simics.

5.1 Replaying Single Thread

We used the Pin Dynamic Instrumentation tool [12] running on Fedora Linux OS to collect the FLLs that were described in Section 4.2 for the programs listed in Table 1. Multiple FLLs were collected per thread leading up to the program crash. The goal of the replayer is to use these FLLs and execute the instructions in the checkpoint interval leading up to the bug. For replaying a program's execution, our replayer has to have access to the exact same binaries for the application and shared libraries used when creating the FLL.

To replay a FLL, we start the program execution under Red Hat Linux simulated by Simics and break the execution before executing the first instruction. We then clear all of the data memory locations, and make sure all of the shared libraries needed for replay are loaded into the virtual address space. We then use the header information in the FLL to initialize the register values and program counter. Finally, the execution of the application is allowed to proceed, breaking before the execution of every load instruction.

On encountering a load instruction, the replayer has to make sure that the memory location accessed by the load contains the right value. To achieve this, it first needs to decompress the next record in the FLL, whose format was described in Section 4.3. Using the bit $LC - type$, we decode the value of $LC - Count$ recorded in the second field of the record in FLL and use it to determine if a load should retrieve its value from the FLL or memory.

If the load value is to come from the log, we use the next full 32-bit value in the log if the $LV - Type$ bit is set. If not, the next 6-bits represent the dictionary entry to provide the value as described in Section 4.3.1. The corresponding dictionary entry is then read and that value is used for the load. Note, to generate the correct values in the dictionary table, we update the dictionary with every executed load and it is simulated during replay exactly the same as described in Section 4.3.1.

Given the above, we can obtain the correct values for the load instructions, and once replay starts, all the instructions executed will update the register file and memory as in normal execution, allowing us to deterministically replay a thread of execution. During replay we will encounter synchronous interrupts, which will be turned into NOPs, since we need not simulate what goes on during an interrupt. To replay past the interrupt, we just continue replaying the next FLL recorded for the thread's execution.

5.2 Replaying Multiple Threads and Inferring Data Races

The individual threads can be replayed in a multi-threaded application using the procedure described in the previous section. But, to debug multi-threaded programs we need to be able to retrieve a valid sequential order of memory operations across all the threads. This is provided by all of the information logged in the MRLs.

For each thread we have the FLLs and MRLs corresponding to multiple checkpoint intervals to be replayed. We can associate the FLL with the MRL that was collected during the same checkpoint interval using the checkpoint interval ID and time stamp stored in the header of these logs. Using the FLL, we generate the trace of instructions executed during the FLL's checkpoint interval, just as in replaying a single threaded application, as described above. This generates an instruction count for each memory operation in each thread, and this is used to map the MRL entries to synchronization points in each thread. Effectively, the MRLs are used to infer the ordering constraints across the threads for the memory operations. Using the checkpoint identifier ($remote.CID$) and the thread ID ($remote.TID$) in the MRL, the checkpoint interval of the remote thread can be identified. Then using the $remote.IC$, we can determine the

Application	Bug Location	Bug Description	Window size
bc 1.06	storage.c line 176	Misuse of bounds variable corrupts heap objects	591
gzip 1.2.4	gzip.c line 1009	1024 byte long input filename overflows global variable	32209
ncompress- 4.2.4	compress42.c line 886	1024 byte long input filename corrupts stack return address	17966
polymorph-0.4.0	polymorph.c lines 193, 200	2048 byte long input filename corrupts stack return address	6208
tar 1.13.25	prepargs.c line 92	Incorrect loop bounds leads to heap object overflow	6634
ghostscript-8.12	ttinterp.c line 5108, ttobjs.c line 279	A dangling pointer results in a memory corruption	18030519
gnuplot-3.7.1	pslatex.trm line 189 plot.c line 622	Null pointer dereference due to not setting a file name A buffer overflow corrupts the stack return address	782 131751
tidy 34132	istack.c at line 31 parser.c at line 3505 parser.c	Null pointer dereference Memory corruption Memory corruption	2537326 13 59
xv-3.10a	xvbmp.c line 168 xvbrowse.c line 956, xvdir.c line 1200	Incorrect bound checking leads to stack buffer overflow A long file name results in a buffer overflow	44557 7543600
gaim-0.82.1	gtkdialogs.c line 759, 820, 862, 901	Buddy list remove operations causes null pointer dereference	74590
napster-1.5.2	nap.c line 1391	Dangling pointer corrupts memory when resizing terminal	189391
python-2.1.1	audioop.c line 939, line 966 sysmodule.c line 76	Arithmetic computation results in buffer overflow A null pointer dereference leads to a crash	92 941
w3m-0.3.2.2	istream.c line 445	Null (obsolete) function pointer dereference causes a crash	79309

Table 1: Open source programs with known bugs. Window size between source of bug and crash is less than a million instructions on an average. The first 5 programs are from the AccMon study, and the rest of the programs are from sourceforge.net. The last set of 4 programs are multithreaded programs.

last committed instruction in the remote thread before the local memory operation was executed. In this way we can deduce the ordering constraints for a memory operation in the local thread relative to the instruction count found in the MRL entry for the remote thread. Once a valid sequential order is retrieved, we know how far to replay each thread before waiting on other threads to get to the same synchronization point.

5.3 Replay Implementation Issues

One issue while replaying is that we need to ensure that the instructions of user and shared library code are loaded into the same virtual addresses as when the FLL was generated. This is required to guarantee that the program counter value recorded in FLL references the right instructions. To provide this information, the operating system driver used to manage the checkpoint logs can be used to also hook the load library routine to record, for a program's execution, where the current starting location is for the binary, each shared library, and user space operating system code. This can be associated with the FLLs for a thread, so that if a bug occurs this binary starting address log can be used by the replayer to correctly set up the code in the virtual address space before starting execution.

A related issue is in replaying self modifying code. Since our checkpoint and memory race logs contain only the loaded data, and not code, code that was modified outside the replay interval cannot be regenerated. One solution would be to also log the first-load of instructions. Another option would be to not support self-modifying code, as is done by many profiling tools in industry.

6 Results

BugNet is based on the principle that the bugs can be reproduced, isolated and fixed by replaying a window of execution corresponding to the moments leading up to the program crash. Though debugging by replaying the program is considered to be an effective technique in the software engineering community [19], there is no clear result on the length of the replay window of execution that is required to capture the majority of the bugs. In this section, we first try to quantify this length, by studying popular desktop applications. We found that replaying 10 million instructions was adequate to characterize a significant number of bugs. Based on this result, we will study the trace sizes and the amount of hardware required in BugNet and also draw a comparison with FDR [24].

6.1 Methodology

To evaluate BugNet, we use a handful of programs from the SPEC 2000 suite to evaluate the online compressor and to analyze the size of the log required for different interval sizes. These include `art`, `bzip`, `crafty`, `gzip`, `mcf`, `parser` and `vpr`. These programs were compiled on x86 platform using -O3 optimizations.

We also provide results for five programs used in the AccMon [26] study, and a handful of other programs that are in the top 100 programs downloaded from the *sourceforge.net* web site. The AccMon programs used are `bc`, `gzip`, `ncompress`, `polymorph`, `tar`. The single threaded sourceforge programs are `ghostscript`, `gnuplot`, `tidy` and `xv`. The multithreaded sourceforge programs are `gaim`, `napster`, `python` and `w3m`.

We make use of Pin [12], an x86 binary rewriting tool, to create the logs examined in this section.

6.2 Bug Characteristics

Table 1 lists the applications with known bugs that we studied. One goal here was to quantify the number of instructions required to be replayed in order to reproduce and fix the bug. The second column in the table gives the details about the location in the source code of the applications which needed to be changed in order to fix the bug. The third column describes the nature of the bug. The fourth column quantifies the length of the replay window required to capture the bug. We determine the size of this window by calculating the number of dynamic instructions executed between the point in the program that was the root cause of the bug and the point where the program crashed.

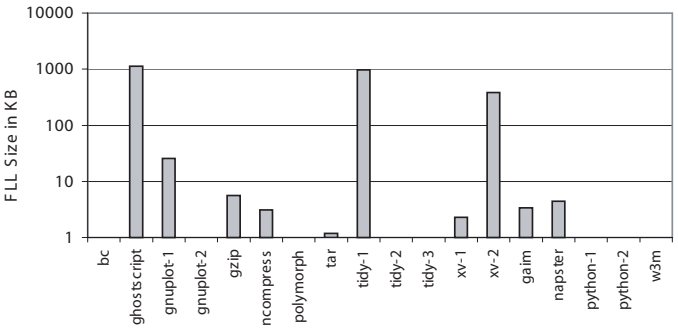

Figure 2: Size of FLLs that can replay the window of execution required to capture the bugs listed in Table 1.

For this analysis, we assumed that the root cause of the bug in the program execution would be the last dynamic instance of the instruction corresponding to the bug fix (the source code location listed in the second column).

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`). It also includes bugs that result in arithmetic overflows (`python`). The worst case in our study is the dangling pointer reference bug found in `ghostscript`. To capture this bug we require a replay window of length 18 million instructions. However, a majority of the other bugs listed in the Table 1 can be captured by having support to replay less than 10 million instructions.

Figure 2 presents the sizes of FLLs that are needed to capture and replay the bugs listed in Table 1. These results assume running BugNet with a 10 million checkpoint interval length. The FLL sizes for several programs are below 1KB, because the number of instructions that need to be replayed to reproduce the bugs in them is only on the order of a few thousand instructions. Except for three applications (`ghostscript`, `tidy` and `xv`), the rest require less than 100KB of FLL information. In the worst case, we require about 1MB of data.

6.3 Sensitivity Analysis

In this section we will discuss how the FLL sizes can vary based upon the checkpoint interval lengths and the replay window lengths. Also, we will study the efficiency of the dictionary based compression algorithm that we discussed in Section 4.3.1. For this study we use SPEC programs, since they have standard inputs, which are well analyzed.

Figure 3 presents the FLL sizes collected for a replay window of 100 million instructions using different checkpoint interval lengths ranging from 10K to 100 million instructions represented along the x-axis. Clearly, as the interval size increases, FLL sizes decrease. This is a result of applying our "first-load" optimization described in Section 4.3. For longer checkpoint interval lengths, it is more probable that a particular memory location referenced by a load has already been recorded and hence the frequency of recording a load instruction decreases resulting in smaller FLL sizes.

Figure 4 shows the sizes of FLLs that are needed to replay a window of 10 million to 1 billion instructions. For these results we assume a constant checkpoint interval length of 10 million

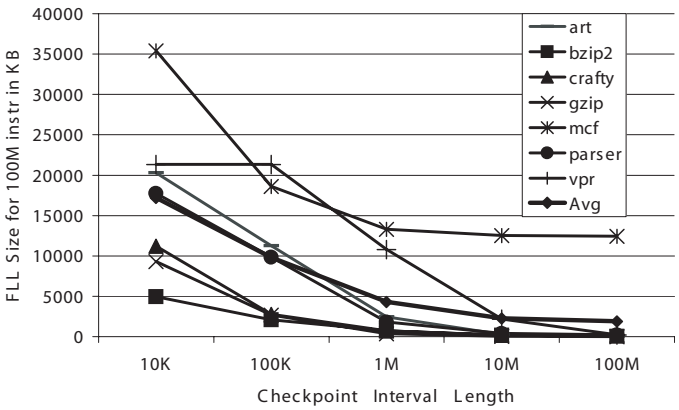

Figure 3: Total size of FLLs required to replay 100 million instructions captured using different checkpoint interval lengths.

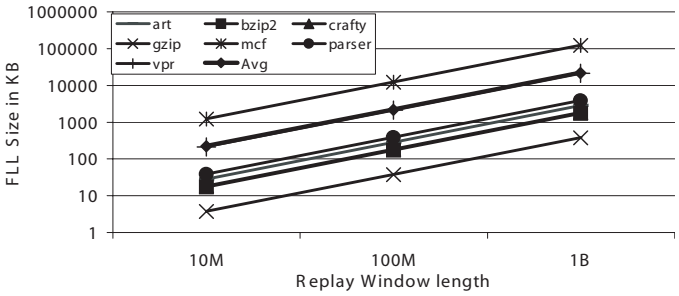

Figure 4: Total size of FLLs required to replay a window of 10 million to 1 billion instructions. FLLs were collected using a 10 million checkpoint interval length.

instructions. On an average, FLLs of size 225 KB are required to replay 10 million instructions and about 18.86 MB for replaying 1 billion instructions.

The results presented so far assume a 64-entry dictionary table for compression. We will now discuss the efficiency of our compression technique described in the Section 4.3.1. Figure 5 shows the percentage of values that were compressible (found in the table) using our dictionary table approach varying the dictionary size. A dictionary of size 64 is capable of compressing 50% of the values on average, which is the size used for the rest of the results in this paper.

Figure 6 shows the compression ratio of FLLs we achieve for various dictionary sizes. On average, we achieve about a 50% compression using a 64-entry dictionary. While a larger dictionary table results in higher compression ratio, it would increase the hardware costs, especially given that the dictionary table is fully associative.

Finally, we used SimpleScalar x86 [2] to examine the performance overhead of BugNet and found it to be less than 0.01% for these SPEC programs. We found for the SPEC programs that the overhead of BugNet is less than 0.01% due to (a) the fact that we use an incremental compression scheme that allows us to lazily write the compressed log entries to memory when the bus is free, and (b) the SPEC programs do not have a lot of interrupts or system calls.

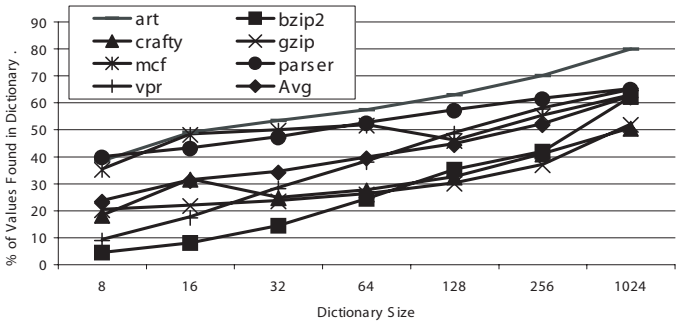

Figure 5: Percent of load values found in the dictionary table of various sizes.

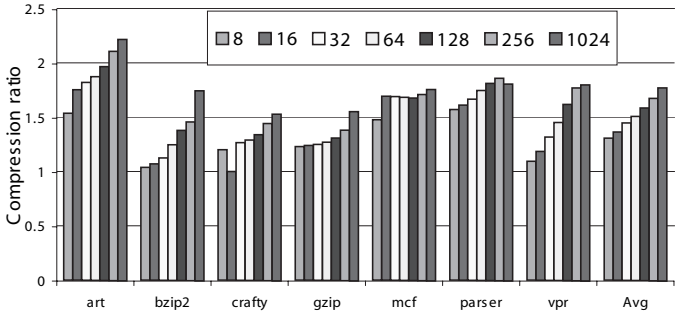

Figure 6: Compression ratios achieved while compressing FLLs using different sizes for the dictionary table. The results are gathered using 10 million checkpoint interval length.

6.4 Complexity of FDR Vs BugNet

FDR's proposal is to have the ability to replay the last 1 second of execution, which can be approximated to a replay window of length one billion instructions, which will vary depending on the processor speed and also the IPC of the program. For a fair comparison with FDR, we discuss using the BugNet architecture to also capture 1 billion instructions. But from the results shown in the Table 1, a replay interval of 10 million instructions should be sufficient to fix many of the bugs in the applications we examined. We therefore also discuss using BugNet to generate logs to replay 10 million instructions.

For the rest of this section all the results presented for BugNet assume a checkpoint interval of size 10 million instructions. Note, a checkpoint interval is different from a replay window. To replay a window of execution, we will use the logs from multiple checkpoints if the checkpoint interval length is less than the desired replay window length. Also on encountering an interrupt we terminate and create a new checkpoint as described in Section 4.4.

6.4.1 Log Size Complexity

Table 2 compares the sizes of BugNet and FDR logs. We compare the amount of memory storage required for replaying 10 million and 1 billion instructions in BugNet. The result for replaying 1 billion instructions in FDR corresponds to replaying one second of execution [24]. If an entry is NIL in the table, then it implies that the log is not used in the mechanism.

The FLL sizes required to replay 10 million and 1 billion instructions are about 225 KB and 18.86 MB on average for the SPEC applications. This assumes a checkpoint interval length of 10 million. In addition, to debug data races we will require memory race logs which should be roughly of the same size as in FDR.

Log size BugNet Vs FDR			
	BugNet:10M	BugNet:1B	FDR:1B
FLL	225 KB	18.86 MB	NIL
Memory Race log	=FDR	=FDR	2 MB
Cache Chk-pnt Log	NIL	NIL	3 MB
Mem Chk-pnt log	NIL	NIL	15 MB
Core Dump	NIL	NIL	128MB-1 GB
Interrupt Log	NIL	NIL	Depends
Prg I/O Log	NIL	NIL	Depends
DMA Log	NIL	NIL	Depends

Table 2: Comparison of log sizes in FDR and BugNet. Interrupt, Program I/O and DMA log sizes will depend on the characteristic of the program. I/O intensive applications will require large sizes for these logs.

Hardware Complexity: BugNet Vs FDR			
	BugNet:10M	BugNet:1B	FDR:1B
CB	16 KB	16 KB	NIL
MRB	32 KB	32 KB	32 KB
Compression	64-entry CAM	64-entry CAM	LZ HW
Chk-pnt Interval	10M instr	10M instr	1/3 sec.
Cache Chk-pnt Buf	NIL	NIL	1024 KB
Mem Chk-pnt Buf	NIL	NIL	256 KB
Interrupt Buffer	NIL	NIL	64 KB
Input Buffer	NIL	NIL	8 KB
DMA Buffer	NIL	NIL	32 KB
Total HW Area	48 KB	48 KB	1416 KB

Table 3: Comparison of hardware complexity in FDR and BugNet. For BugNet we consider support for replaying 10 million instructions as it is adequate to capture the replay window for most of the programs in Table 1. Hardware complexity is also shown for BugNet to capture a 1 billion replay window, which is the window size assumed for FDR.

FDR to support replaying 1 billion instructions would require 18 MB of cache and memory logs as described in [24], plus memory race logs of size 2 MB. The combined size of these is roughly the same as the sizes of using FLLs for capturing 1 billion instructions. However, FDR requires additional information to enable full system replay. FDR records Interrupt, I/O and DMA logs whose sizes may vary widely depending on the nature of the application. For I/O intensive applications, these logs might have prohibitive sizes. In addition, FDR requires a core dump image whose size can range up-to 1GB, based on the application's memory footprint and the main memory size.

Our results show that a log of size 225 KB can replay 10 million instructions of an application's execution. This should be enough to reproduce and debug a majority of the bugs, at least for the program's we examined. In addition, BugNet's small traces (sometimes on the order of only hundreds of KB) should encourage users to communicate the logs back to the developer.

6.4.2 Hardware Complexity

Table 3 compares the hardware complexity of BugNet and FDR [24]. Like in the previous section, here again we compare the configuration of BugNet to capture 10 million and 1 billion instructions.

The main hardware structures used in BugNet are the CB, MRB hardware buffers and a fully associative 64-entry dictionary table as shown in the Figure 1. The size of the CB needs to be only large enough to tolerate bursts in our logging. In addition, we perform incremental compression of each log entry, which allows us to lazily write the logs into main memory and free up space in the CB. The sizes of the CB, MRB and dictionary table will be a constant irrespective of the length of replay window that we are trying to capture, since the logs are memory backed.

In comparison, FDR requires about 1416 KB of on-chip hardware to record enough information for full system replay. FDR assumes hardware implementation of LZ [28] compression. The LZ compressor is block-based, so the hardware buffer size needs to be large enough to collect a block of information before compressing and storing it back to main memory, and it also needs to be large enough to tolerate bursts. Cache and Memory checkpoint buffers are used to record information required by the SafetyNet checkpoint mechanism, whose sizes are 1 MB and 256 KB respectively. Since FDR aims to achieve full system replay it has to record all the external inputs for which it requires three additional buffers - 64 KB interrupt buffer to record interrupts, 8 KB input buffer to record program I/O and a 32 KB DMA buffer to record DMA writes. In summary, the total on-chip hardware requirement for BugNet is about 48 KB, whereas FDR requires 1416 KB.

7 Limitations

In this section we will discuss the limitations of our BugNet architecture in providing support to capture bugs in released software. The primary limitation is that bugs that occur due to the complex interactions of the application program and the system code will be difficult to fix, since we track only the application code. The other limitation is that the amount of memory space devoted for BugNet is limited, and bugs that require a large replay window may not be captured.

7.1 Debugging

The focus of BugNet is to assist debugging user code that does not have complex interactions with operating system routines like drivers and interrupt handlers. 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 after servicing the interrupt. This, along with the 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. The user code along with the OS library code consist of a significant portion of the program's execution, and this should be sufficient to track down the majority of application-level bugs.

Note that BugNet logs do not contain a core dump representing the final state of the entire system/main memory. As a result, one cannot examine arbitrary memory locations or data structures. If the memory location was not accessed during the instruction window being replayed, its value cannot be examined during replaying. This might cause slight inconvenience in inferring the cause of a bug, but it should not prevent the user from isolating the bug, since we expect that the memory addresses untouched by the program's execution prior to the crash were not responsible for the faulty behavior.

BugNet can detect a bug only when the operating system or the application itself can identify that the program has encountered a fault or exception. For example, bugs resulting in incorrect results would not be captured through BugNet, since it does not know the right time to capture the bug.

7.2 Replay Window Size

Another potential issue is the replay window size, if it turned out to be too small to capture the bug. The replay window size is a knob that can be tuned by the operating system or the user. The replay window size is essentially dependent on the amount of main memory space that can be allocated for storing all of the FLLs and MRLs used to capture the desired number of instructions for replay. The user should be able to specify a desired replay window size and the maximum performance penalty that they are willing to pay for it. Based on this input, the operating system can dynamically tune the memory space allocation. If the OS can determine that the applications running at a particular instant of time are not memory intensive and that considerable amount of free space is available, then it can increase the memory space allocation to BugNet. On the other hand, if the performance degradation goes above the tolerable limits it can tune down the space allocated. In addition, the customer using the application can specify a minimal replay window size that can capture a majority of the bugs. We found that a replay window of size 10 million instructions is enough to capture the majority of bugs. If a bug occurred, but not enough state was kept to track down the bug, then the customer may be asked to increase the replay window size.

8 Conclusion

The computer industry has long realized the fact that released software is bound to contain bugs. One of the key challenges faced today by the software developers is in reproducing the bugs that manifest themselves at the customer site. To address this problem, we proposed the BugNet architecture that continuously records information during production runs. Recorded information (less than 1MB) can be communicated back to the developer and used by them to characterize the bug by deterministically replaying the program's execution before the crash.

BugNet focuses on replaying only user code and shared libraries to find application level bugs. To achieve this, BugNet's logs for a checkpoint interval contain the register state at the start of the interval and a trace of first load memory accesses. This is enough information to achieve deterministic replay of a program's execution, without having to replay what goes on during interrupts and system calls. This results in small log sizes, where a log size of around 500KB is enough to capture a replay window size of 10 million instructions. This is small enough

to motivate users to communicate the log back to the developer. We found that BugNet has very little performance overhead, and the area overhead is around 48 KB for the few hardware buffers required.

Acknowledgments

We would like thank Yuanyuan Zhou and her students for providing us with a set of programs with known errors and inputs that expose them. We would also like to thank Jack Sampson and the anonymous reviewers for providing useful comments on this paper. This work was funded in part by grants from ST Microelectronics, Intel and Microsoft.

References

[1] 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.

[2] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[3] M. Burtscher and N. B. Sam. Automatic generation of high-performance trace compressors. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 229–240, Washington, DC, USA, 2005. IEEE Computer Society.

[4] G. Candea. Enemies of dependability software (lecture notes: Cs444a).

[5] 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.

[6] M. L. Corliss, E. C. Lewis, and A. Roth. Low-overhead interactive debugging via dynamic instrumentation with dise. In *Proceedings of 11th International Symposium on High-Performance Computer Architecture*, Feb 2005.

[7] Netscape Communications Corp. Netscape quality feedback system. *http://wp.netscape.com/*.

[8] Microsoft Corporation. Dr. watson overview. *http://oca.microsoft.com/en/dcp20.asp*.

[9] W. R. Dieter and J. E. Lumpp Jr. A user level checkpointing library for posix threads programs. In *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, page 224. IEEE Computer Society, 1999.

[10] E. N. Elnozahy, L. Alvisi, Y. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Survey*, 34(3):375–408, 2002.

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

[12] 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.

[13] S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, 2002.

[14] 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.

[15] M. Prvulovic and J. Torrelas. Reenact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *30th Annual International Symposium on Computer Architecture*, San Diego, CA, June 2003.

[16] M. Prvulovic, Z. Zhang, and J. Torrellas. Revive: Cost effective architectural support for rollback recovery in shared-memory multiprocessors. In *Proceedings of the 29th Annual International Symposium on Computer architecture*, pages 111–122. IEEE Computer Society, 2002.

[17] F. Qin, S. Lu, and Y. Zhou. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *Eighth International Symposium on High Performance Computer Architecture*, February 2005.

[18] M. Ronsse and K. De Bosschere. Non-intrusive on-the-fly data race detection using execution replay. In *Proceedings of Automated and Algorithmic Debugging*, Nov 2000.

[19] M. Ronsse and K. De Bosschere. Debugging backwards in time. *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG)*, Sep 2003.

[20] 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.

[21] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. Safetynet: Improving the availability of shared-memory multiprocessors with global checkpoint/recovery. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 123–134. IEEE Computer Society, 2002.

[22] 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.

[23] Geodesic Systems. Geodesic traceback - application fault management monitor., 2003.

[24] 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.

[25] J. Yang and R. Gupta. Energy efficient frequent value data cache design. In *IEEE/ACM 35th International Symposium on Microarchitecture*, pages 197–207, 2002.

[26] 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.

[27] P. Zhou, F. Qing, W. Liu, Y. Zhou, and J. Torrellas. iwatcher: Efficient architecture support for software debugging. In *31st Annual International Symposium on Computer Architecture*, June 2004.

[28] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.