A Case for an Interleaving Constrained Shared-Memory Multi-Processor

Jie Yu
University of Michigan, Ann Arbor
jieyu@umich.edu

Satish Narayanasamy
University of Michigan, Ann Arbor
nsatish@umich.edu

ABSTRACT

Shared-memory multi-threaded programming is inherently more difficult than single-threaded programming. The main source of complexity is that, the threads of an application can interleave in so many different ways. To ensure correctness, a programmer has to test all possible thread interleavings, which, however, is impractical.

Many rare thread interleavings remain untested in production systems, and they are the root cause for a majority of concurrency bugs. We propose a shared-memory multiprocessor design that avoids untested interleavings to improve the correctness of a multi-threaded program. Since untested interleavings tend to occur infrequently at runtime, the performance cost of avoiding them is not high.

We propose to encode the set of tested correct interleavings in a program's binary executable using *Predecessor Set (PSet)* constraints. These constraints are efficiently enforced at runtime using processor support, which ensures that the runtime follows a tested interleaving. We analyze several bugs in open source applications such as MySQL, Apache, Mozilla, etc., and show that, by enforcing PSet constraints, we can avoid not only data races and atomicity violations, but also other forms of concurrency bugs.

Categories and Subject Descriptors

B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Design, Reliability, Verification

Keywords

Multiprocessors, Parallel Programming, Concurrency Bugs, Software Reliability

ISCA'09, June 20–24, 2009, Austin, Texas, USA.

1. INTRODUCTION

Parallel programming has remained a very challenging problem for the last several decades. In the explicit parallel programming model, a programmer discovers parallelism in the application, and explicitly specifies independent parallel tasks. Explicit parallel programming is error-prone and inherently harder than von Neumann style single-threaded programming. To understand its underlying complexity, consider the task of program verification. Verifying even a single-threaded program is NP-complete (because of pointers, loops, etc.). However, this problem can be decomposed into a set of contracts (e.g.: loop-invariants, pre-conditions and post-conditions for functions) that are verifiable in polynomial time for a single-threaded program. Traditionally, these contracts are what the programmers manually try to understand, test and verify to ensure correctness. However, in a shared-memory multi-threaded program, the problem of verifying even these contracts is NP-complete [19]. The reason is that, in a shared-memory multi-threaded program, the memory operations executed by a thread's function or loop could *interleave* with the memory operations executed by all the other threads in so many different orders. As a result, the number of legal states to verify at each program statement increases exponentially, making the task of verifying even simple contracts for functions and loops NP-complete.

A programmer could use synchronization operations such as semaphores, locks, condition variables, transactions, etc., to reduce the number of legal thread interleavings. A stronger memory consistency model would also reduce the number of legal thread interleavings. Even so, the interleaving space of most multi-threaded programs is so large that, a programmer cannot practically test all the legal interleavings of a program. Thus, one of the fundamental problems with the shared-memory multi-threaded programming model is that, it exposes too many legal interleavings to the parallel runtime system, which makes it difficult for programmers to guarantee correctness.

Even in a well tested production code, a programmer would have tested most of the frequently occurring interleavings, but many of the infrequently occurring interleavings would remain untested. These rare untested interleavings are the root cause of a majority of concurrency bugs in the production code.

This paper proposes to constrain a shared-memory multiprocessor system, such that it avoids the untested thread interleavings during a production run. This improves correctness, because a tested interleaving is more likely to be correct than an untested interleaving. Also, the untested in-

terleavings tend to occur infrequently during an execution, because otherwise they would have been tested in a high quality production code. Therefore, constraining a parallel runtime system to avoid the rare untested interleavings does not degrade performance significantly.

Two key solutions are discussed in this paper. One, we propose to encode the set of tested correct interleavings in a program's binary executable using *Predecessor Set (PSet)* constraints. Two, we discuss processor support for efficiently enforcing the PSet interleaving constraints in order to avoid untested interleavings during a production run.

The first challenge is to develop a method for encoding the set of all tested thread interleavings in a program's executable using ISA extensions. We define a thread interleaving of an execution to be the order between the memory operations executed by all the threads. A thread interleaving is therefore unique to an execution of a program for a particular input. The challenge is to derive interleaving constraints from each tested interleaving. These interleaving constraints should be generic enough for different program inputs, so that enforcing them does not result in too many unnecessary constraint violations during the production runs. At the same time, the interleaving constraints should also be able to capture the set of tested interleavings, so that by enforcing them, we can avoid a majority of concurrency bugs due to the untested interleavings.

This paper presents an interleaving constraint called the *Predecessor Set* (PSet) constraint, which meets the above two requirements. Every memory instruction in the program's binary has a PSet constraint defined for it. The PSet constraint for an instruction I specifies the set of all valid remote memory operations over which I can be immediately dependent upon. Intuitively, PSet constraints captures the tested interleavings between two dependent memory instructions. If a memory operation I was never immediately dependent upon another remote memory operation P in any of the test runs, then that dependency is avoided during the production runs. We show that by enforcing PSet interleaving constraints, we can effectively avoid not only bugs due to data races and atomicity violations, but also other forms of concurrency bugs that cannot be found using the existing dynamic bug detection tools.

The second challenge is to efficiently enforce PSet constraints during production runs in order to avoid untested interleavings. We discuss extensions to a shared-memory multi-processor design, which efficiently detects PSet constraint violations and avoids them. Without processor support, we find that the performance overhead for enforcing the PSet constraints is very high, which would render the proposed interleaving constrained execution model to be impractical.

Our interleaving constrained processor detects PSet violations by keeping track of the last accessor information for every memory word and piggy-backing the coherence messages with that additional information. The processor recovers from a PSet constraint violation by either stalling the violating thread until the constraint is satisfied, or by re-executing the program from an earlier checkpoint with an alternative thread interleaving. The violated constraint is logged and sent back to the developer. The developer can then test the violated interleaving seen in the production run. If it is indeed a correct interleaving, then a binary patch to relax the relevant PSet constraint could be released.

We discuss a software tool built using Pin [13], which we use to profile applications during the test runs and derive the PSet constraints. It is also used to detect and avoid the PSet constraint violations during real executions. This tool is used to analyze the effectiveness of the proposed mechanism in avoiding several bugs in the open source programs such as MySQL, Mozilla, Apache, etc. We also analyze the accuracy and performance of our tool using Splash [24] and Parsec [2] benchmark suits.

This paper makes the following contributions:

- It is impractical for a programmer to test all the legal interleavings in a multi-threaded program. Untested interleavings are the root cause for a majority of concurrency bugs. Therefore, to improve correctness, this paper makes a case for a mechanism that constrains a production run to follow a tested interleaving.
- We encode the set of correct tested interleavings using an interleaving constraint called the Predecessor Set (PSet). We show that, by enforcing the PSet constraints during production runs, we can effectively avoid data races, atomicity violations, and also other forms of concurrency bugs that cannot be found by existing dynamic analysis tools. This analysis is based on 17 real bugs in open source applications such as Apache, Mozilla, MySQL, etc.
- We present a shared-memory multi-processor design that efficiently detects and avoids PSet constraint violations. It overcomes the violated PSet constraints either by stalling the violating thread, or by re-executing the program from an earlier checkpoint with an alternative interleaving. We show that the number of PSet constraint violations in a bug free execution is very less, and as a result, the performance cost of enforcing the PSet constraints is negligible.

2. BACKGROUND AND PRIOR WORK

The premise of this paper is that it is impractical for programmers to ensure the correctness of all the legal interleavings in a shared-memory multi-threaded program. Instead of letting the runtime execute any of the legal interleavings, including the untested interleavings, we propose to constrain the runtime to follow a tested interleaving, whenever possible. We show that this interleaving constrained execution model could improve the correctness of multi-threaded programs.

A key component of the proposed model is the method to derive interleaving constraints from the test runs and encode those constraints in a program's binary. The derived interleaving constraints should be such that, when they are enforced during a production run, it should result in an interleaving that is similar to a tested interleaving. We discuss Predecessor Set (PSet) constraints that meet this goal.

By enforcing the PSet constraints and thereby mimicking a tested interleaving, we can avoid a majority of concurrency bugs. Instead of using PSet constraints to avoid concurrency bugs, one could enforce the invariants used in existing concurrency bug detection tools such as data race detectors [21, 14] and/or atomicity violation detectors [25, 11, 12]. On detecting a violation of these invariants, the runtime system can recover from them using a checkpoint, rollback and re-execution mechanism [17, 20].

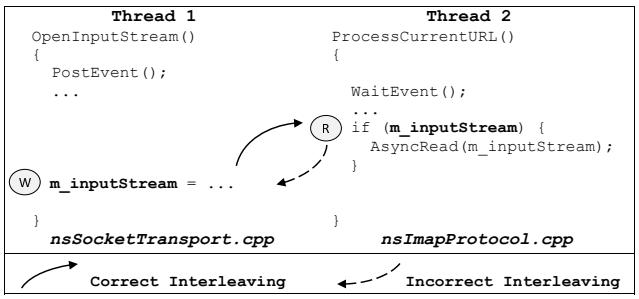

Figure 1: A data race bug in Mozilla (Bug #13 in Table 1).

This section discusses the limitations of two invariants (data races and atomicity violations) commonly used in dynamic concurrency bug detection tools, and illustrate why they are not generic enough for avoiding a wide range of concurrency bugs in a production system. A fundamental difference between the PSet constraints and the existing invariants is that, the PSet constraints are designed to detect the untested interleavings, whereas the existing invariants are designed to detect interleaving patterns that have a strong correlation with the mistakes that the programmers frequently make in a multi-threaded program. In Section 5, we compare the effectiveness of PSets with a happens-before data race detector [14] and an atomicity violation detector [11] in avoiding concurrency bugs.

2.1 Data Race Detectors

A data race can be defined as a pair of memory accesses to the same memory location, where at least one of the accesses is a write, and neither one happens-before the other. Dynamic data race detectors can be classified into two major categories: happens-before based and lockset based. A happens-before data race detector [15] finds only the data races that manifest in a given program execution. Lockset based techniques [21] can predict data races that have not manifested in a given program execution, but can report false positives. If the goal is to avoid the concurrency bugs in a production run, then a happens-before data race detector is a better candidate, as it finds only the true data races that occur in a given program's execution. A lockset-based data race detector [21], on the other hand, will find data races that do not occur in a program's execution, which is useful for detecting more bugs, but not so useful detecting and avoiding bugs at runtime.

Figure 1 shows a data race bug in Mozilla. In this example, the programmer intended to execute the write `W` before the read `R`. This is likely to happen most of the time, in which case, the read `R` would get the right value. However, this code lacks proper synchronization to guarantee the required happens-before relation. As a result, in some execution it is possible that `R` executes before `W`. This would result in an incorrect execution, where an input to the program would be left unprocessed. A happens-before based data race detector can detect this incorrect interleaving. On detecting a data race, the runtime can re-execute the program with an altered interleaving using a checkpoint and rollback mechanism. ReEnact [17] and Rx [20] used such an approach.

However, not all data races are harmful data races [14].

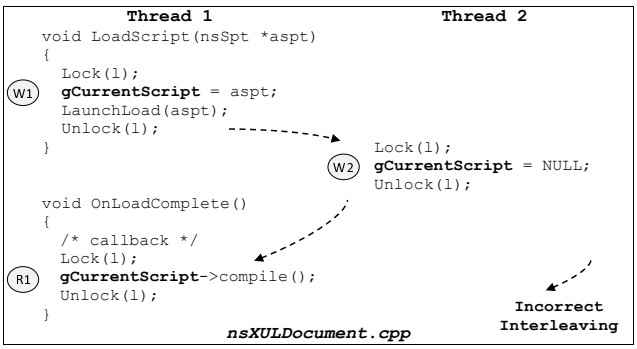

Figure 2: An atomicity violation bug in Mozilla [11] (Bug #7 in Table 1).

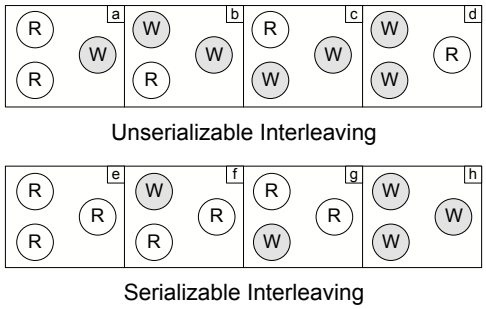

Figure 3: Unserializable and serializable memory interleavings used in AVIO [11] to detect atomicity violations.

Many of the data races in production systems are benign, as programmers intentionally allow data races to optimize performance. Programmer constructed synchronizations could also result in benign data races. Benign data races also tends to be frequent, as opposed to harmful data races [14]. Thus, triggering a rollback and recovery on every data race violation might cause significant performance overhead. In Section 3, we discuss how a mechanism based on PSet constraints correctly permits benign data races at runtime.

More importantly, a data race detector cannot detect a number of concurrency bugs. A data race is really a heuristic, which is used to detect a particular interleaving pattern (concurrent memory accesses with no happens-before relation between them) that strongly correlates with the concurrency bugs that programmers tend to make. Presence of a happens-before relation between two concurrent memory operations does not necessarily imply that the order between the two memory accesses is correct. A good counter-example is an atomicity violation bug shown in Figure 2. In this example, `W1` and `R1` is expected to execute atomically. But, `W2` can get interleaved between the two, leading to an incorrect execution. A data race detector cannot detect this concurrency bug, because there is a happens-before relation between all the accesses to the variable `gCurrentScript`.

2.2 Atomicity Violation Detectors

There have been work on detecting atomicity violations [25, 11, 5]. Most of the atomicity violation detectors rely on programmers to specify the atomic regions through annotations. Using these annotations, static analysis tools can

detect potential atomicity violations in a program [5]. Alternatively, atomicity for the user specified atomic regions can be ensured at runtime using transactional memory [7, 6]. However, programmers might incorrectly specify smaller atomic regions, which could lead to concurrency bugs. Even if a programmer manages to correctly enforce atomicity for the required code regions, incorrect interleavings between atomic regions could lead to incorrect executions. The PSet constraints discussed in Section 3 avoids atomicity violations without requiring programmer annotations, and also avoids the incorrect interleavings between atomic regions, as the PSet constraints are learned from the test runs.

SVD [25] and AVIO [11, 9] propose heuristics to automatically infer atomic regions and detect atomicity violations using dynamic analysis. AVIO [11, 9] is related work to our work in that the atomicity constraints in AVIO are learned by profiling program executions, similar to how we derive PSet constraints. Here we describe AVIO in more detail. We also provide detailed results comparing PSet constraints to AVIO in Section 5.

AVIO [11] infers atomicity invariants from the training runs. An atomicity invariant consists of a pair static memory instructions. When a thread executes the two memory operations of an invariant pair, they cannot interleave with an *unserializable* memory operation accessing the same location in a different thread. Figure 3 shows the serializable and unserializable interleavings.

The atomicity violation bug shown in Figure 2, which we discussed earlier, can be detected by AVIO. This is because, in all the correct training runs, `W1` and `R1` would have executed atomically. In any other execution, when AVIO is turned on, if `W2` is interleaved between the invariant pair, then a violation would be detected. Thus, AVIO is simple, and is also effective in detecting a number atomicity violations.

However, AVIO cannot detect all of the atomic violations. Let us consider a bug in Mozilla shown in Figure 4. The memory operations `W1`, `R1`, and `W2` should all be executed atomically. Otherwise, the function `HandleEvent` executed in the second thread would return without processing an event. AVIO cannot detect this bug, because `R2` can validly interleave between `W1-R1` and `R1-W2` without violating any AVIO invariant, because both of them are serializable interleavings. Section 3 describes how bugs like the one shown in Figure 4 are detected and avoided using the PSet constraints. In fact, the proposed PSet constraints are a superset of the constraints specified by the AVIO invariants. Also, unlike AVIO [11], the goal of this paper is not only to detect concurrency bugs during testing, but also to avoid them in production systems.

2.3 Motivation for PSet Constraints

ReEnact [17] and Rx [20] detect data races at runtime and avoid them using a checkpoint, rollback, and re-execution mechanism. Atom-Aid [12] is an architectural mechanism that arbitrarily groups instructions together to create atomic blocks, and thereby probabilistically reduces the chances for an atomicity violation during production runs. All these mechanisms target a sub-class of concurrency bugs. In contrast, a PSet constraint based mechanism can avoid data races, atomicity violations, and also other forms of concurrency bugs. Figure 5 shows a bug that is neither a data race nor an atomicity violation. The function `Notify()` in

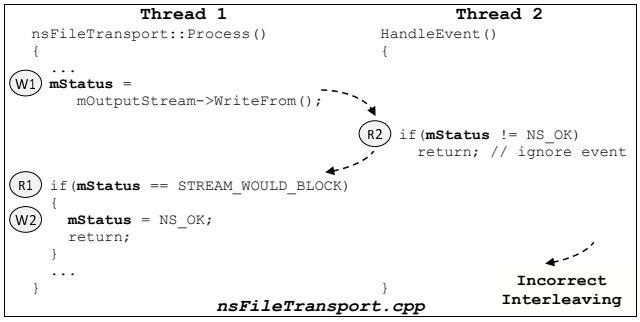

Figure 4: An atomicity violation bug in Mozilla, which will not raise an AVIO [11] invariant violation. (`Bug #10` in Table 1).

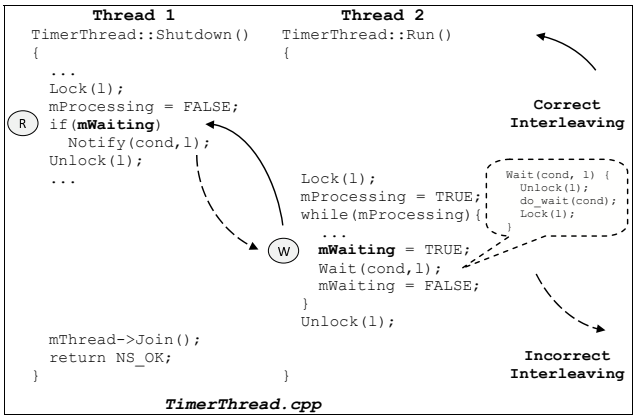

Figure 5: Order violation bug in Mozilla, which is neither a data race nor an atomicity violation. (`Bug #15` in Table 1).

`Thread1` should be invoked only after `Thread2` executes the `Wait()` function. Otherwise, `Thread2` would block forever. Lu et al. [10] analyzed several concurrency bugs, and they also note that there are number of bugs that are neither data races nor atomicity violations. The proposed PSet constraints can detect and avoid these bugs as well. Also, avoiding untested interleavings using PSet constraints provides a useful guarantee to the programmer that only the tested interleavings between the memory operations would be executed at runtime. Such an approach could help improve the testing methodology for the multi-threaded programs.

2.4 Deterministic Multi-threading

Deterministic Multi-Processor (DMP) [4] and Kendo [16] are two recent developments. They constrain the thread interleavings to provide a guarantee that any execution of a multi-threaded program would yield the same output as long as the input remains the same. In other words, they guarantee a deterministic order of all memory accesses for a given program input. This could help programmers in reproducing bugs. However, for a given input, the system chooses the deterministic order based on arbitrary program events (for example, number of retired stores). As a result, the chosen deterministic order is not going to be more correct than a random order chosen by the current systems. Therefore, a programmer would still have to test all legal in-

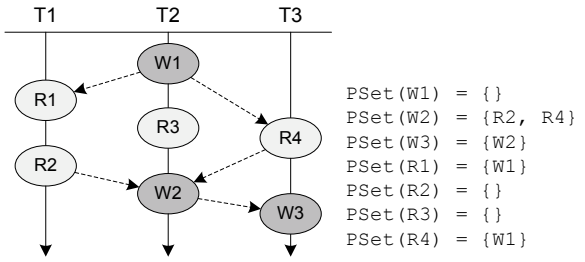

Figure 6: PSet constraints for an interleaving.

terleavings. This is a fundamental problem with the shared-memory multi-threaded programming model, which we seek to address in this paper.

3. ENCODING AND ENFORCING TESTED INTERLEAVINGS

This section discusses the Predecessor Set interleaving constraints. They are derived from correct test runs, and are encoded in the program binary. We then discuss the effectiveness of PSets in avoiding concurrency bugs using the examples discussed in Section 2. We also discuss methods to detect and avoid PSet constraint violations during production runs. Finally, we discuss the limitations of the PSet constraints.

3.1 Predecessor Sets (PSets)

In this paper, we take the first step towards constraining a shared-memory multi-processor system to follow the tested interleavings. We focus on constraining just the interleaving between two dependent memory operations using the *Predecessor Set* constraints. We show that even with this simplification, the PSet constraints are powerful enough to avoid most of the data race bugs, atomicity violation bugs, and also other concurrency bugs.

A *Predecessor Set (PSet)* is defined for each static memory operation. The PSet for a memory instruction specifies the set of all memory operations over which it can be immediately dependent upon. We consider true (read-after-write) as well as false (write-after-write and write-after-read) dependencies. We consider all thread local memory dependencies (where the two dependent memory operations were executed in the same thread) to be valid during production runs. Therefore, a PSet constraint specifies only the set of valid remote dependencies for an instruction.

A static memory operation M contains another static memory operation P in its PSet only if the following conditions are satisfied. Either P or M should be a write. Also, there should be at least one dynamic instance in any of the tested correct interleavings, such that (a) P and M were executed in two different threads (say, `T1` and `T2`), (b) M was immediately dependent on P in that interleaving, and (c) neither `T1` nor `T2` executed a read or a write (to the same memory location as M) that interleaved between P and M. During a production run, the runtime system detects a violation while executing a memory operation M, if M is memory-dependent on a remote memory operation P, but P is not in the PSet of M.

Figure 6 shows a tested interleaving, and the resultant PSets. Assume that all the memory operations shown in the figure are to the same memory location, and each of them is a different static memory operation. Reads are labeled with the prefix `R` and writes are labeled with the prefix `W`. The PSet for `W2` contains two reads due to two write-after-read dependencies. Note that the PSet for `R2` is empty even though it is immediately dependent on the remote memory operation `W1`, because `R1` interleaves between `W1` and `R2` (refer condition (c) listed above). In this tested interleaving, the values read by `R1` and `R2` would be the same. An empty PSet for `R2` ensures that the value read by `R1` and `R2` are the same even in the production runs. Including `W1` in the PSet of `R2` would not guarantee this property during production runs.

3.2 Effectiveness of PSets in Avoiding Concurrency Bugs

Using the examples discussed in Section 2, we now describe how enforcing the PSet constraints avoids harmful data race bugs (while still allowing benign data races for performance), atomicity violations and other forms of concurrency bugs.

3.2.1 Enforcing PSet Constraints Avoids Data Races

Figure 1 shows a data race bug in Mozilla. In this example, the variable `m_inputStream` points to a heap location that is dynamically allocated on receiving an input. During the correct test runs, the only valid interleaving between `W` and `R` is `W` $\rightarrow$ `R`. Therefore, the PSet for `R` contains `W`, and the PSet for `W` is a null set. For this data race bug to manifest in a production run, `R` should precede `W`. However, such an interleaving would result in at least one PSet constraint violation. The predecessor for `W` in an incorrect interleaving would be `R`. Since `R` is not in the PSet for `W`, a PSet constraint violation would be detected.

We discussed about benign data races [14] in Section 2. A benign data race could occur frequently in a program's execution. Therefore, if we use a data race detector to avoid data races at runtime, we might hurt performance. However, a PSet constraint based mechanism does not have this issue, as PSets can capture the fact that a benign data race interleaving is a correct interleaving, provided that interleaving is seen in a correct test run.

3.2.2 Enforcing PSet Constraints Avoids Atomicity Violations

Figure 2 shows an atomicity violation bug in Mozilla. The memory operations `W1-R1` are expected to execute atomically. `W2` would never be immediately dependent on `W1` in any of the correct test runs. Therefore, the PSet for `W2` would not contain `W1`. In an incorrect execution, the atomicity property of `W1-R1` could be violated by an interleaving `W2`. However, this would cause a PSet constraint violation at `W2`, as the PSet of `W2` would not contain `W1`.

For this example, we would detect a PSet the violation at `W2`, whereas AVIO can only detect the violation later at `R1`. A PSet constraint violation can be detected at least as early as an AVIO constraint violation, as the PSet constraints are a super-set of the AVIO constraints.

Figure 4 shows an atomicity violation bug that AVIO [11] cannot detect. The programmer expects that the operations `W1-R1-W2` be executed atomically. Therefore, the PSet for `R2` learned from all the correct test runs would not contain `W1`. When the required atomicity property for the operations `W1-`

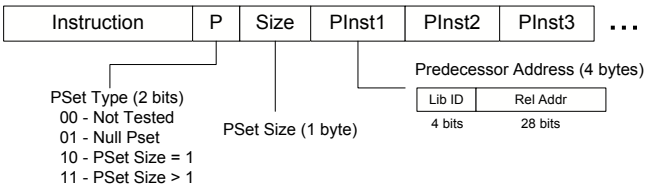

Figure 7: Format for encoding an instruction's PSet.

`R1-W2` is violated by `R2` in a production run, a PSet violation would be detected at `R2`.

3.2.3 Enforcing PSet Constraints Avoids Order Violations

Figure 5 shows a concurrency bug in Mozilla that neither a data race detector nor an atomicity violation detector can detect. We described this bug in Section 2.3. The PSet for `W` learned from the correct test interleavings would not contain `R`. In an incorrect interleaving during a production run, `R` would be `W`'s predecessor, and therefore a PSet constraint violation would be detected at `W`.

3.3 Deriving and Encoding PSets Constraints

The predecessor sets are constructed from the test runs using a profiling tool that we built using Pin [13]. The programmer has to ensure that the test run is correct. This could be done by verifying the program output and by checking the test run using dynamic bug detection tools.

Figure 7 shows the format of an instruction with its PSet information for a 32-bit ISA. The field P-Type has two bits. If the P-Type value for an instruction is three, then the next field specifies the number of instructions in the PSet for that instruction. Each of the remaining fields specify an instruction in the PSet. An instruction is represented using a concatenated value of the identifier for its library and its relative offset that refers to the instruction's location in the library. This is necessary to support programs with dynamically loaded libraries.

The worst case space complexity for the PSets of a program is $O(N^2)$, where `N` is the number of static memory instructions in the program. The reason is that, each static instruction can have at most `N` elements in its PSet. However, in Section 5 we show that, on average, about 95% of static instructions have a PSet of size zero, as a huge proportion of memory operations are thread local accesses. For such instructions, there is no additional space overhead.

Programmers commonly use a testing metric called basic block coverage. It measures the percentage of static instructions that were executed in at least one test run. Even for high quality production code, basic block coverage is typically less than 100%. For instructions that were never tested even once, we could assume that its PSet is a null-set. This could ensure a high degree of fault tolerance. Alternatively, one could choose not to enforce PSet constraints for such untested instructions to reduce the number of false constraint violations during production runs. However, untested instructions are also likely to occur rarely, because otherwise it would have been tested in a well tested program. Therefore, assuming null PSets for untested instructions in a well tested program would not result in significant number of false constraint violations.

3.4 Detecting and Enforcing PSet Constraints

We now discuss methods to detect and avoid PSet constraint violations. Using these methods we ensure that most of the untested interleavings are avoided during production runs. An architectural support for detecting and avoiding PSet constraint violations is discussed in Section 4).

During a production run, whenever a memory operation is immediately dependent on a remote memory operation, the runtime system checks to see if the remote memory operation is in the predecessor set of the current memory operation. If not, a PSet violation is detected.

To repair the violation, we evaluate two approaches. In one approach, the violating memory operation is stalled until the violation gets resolved. When the violating thread is stalled, other threads continue to make progress. If another thread executes a memory operation to the same memory location as the violating memory operation, the violated PSet constraint is checked again. If the check succeeds, the stalled thread continues its execution.

A repair mechanism based on stalling the violating threads is easier to support and is also performance efficient. However, not all PSet constraint violations can be avoided using this mechanism. Because, for a constraint to get resolved, another thread should be able to make progress so that it eventually accesses the same memory location as the stalled memory operation. But, it is possible that the other thread needs a lock before it can access that memory location, and that lock might be currently held by the stalled thread. Thus, stalling the violating thread might not resolve the violation. Consider another example where a violating memory operation's PSet is a null-set. In this case, any remote memory dependency would cause a violation, and therefore waiting for the other threads to execute a different memory operation is never going to resolve the violation. To ensure forward progress while using a stalling mechanism, we use a time-out scheme, where the stalled thread is released to continue its execution (or the second recovery scheme is triggered, if available) when the stall time has reached a particular threshold.

We also evaluate another recovery mechanism to avoid PSet constraint violations. It is based on a checkpoint and rollback mechanism. On detecting a violation, the program is re-executed from an earlier checkpoint. During re-execution, the thread schedule is perturbed to induce an alternative interleaving. Since, the constraint violations are likely to be rare events, it is unlikely that the same violation would be encountered again during re-execution.

Not all PSet constraint violations can be avoided by just perturbing the thread schedule. It is possible that the only legal interleaving for an input is one that is untested. Such violations would cause repeated rollbacks to the same checkpoint. To ensure forward progress, the maximum number of rollbacks to a checkpoint is set to a threshold. When the number of rollbacks to a checkpoint has reached the threshold, that checkpoint is discarded, and another checkpoint is taken at the point where a PSet violation is detected. The system then logs the violation and continues with the execution. This ensures forward progress. The log is sent back to the developer to test the untested interleaving and determine if it is a cause of a bug or not. If it is not a bug, then the relevant PSet is updated in order to allow the newly tested interleaving at runtime.

3.5 Limitations

The PSet constraints described in this paper does not account for the interleavings between two or more memory operations accessing different memory locations. As a result, it may not be able to avoid certain bugs due to multi-variable atomicity violations. Another limitation of PSets is that they are context insensitive. Additional context such as the calling stack could help avoid more untested interleavings. In future, we plan to extend the PSet interleaving constraints so that we can avoid most of the untested interleavings. However, our analysis in Section 5 shows that the PSets constraints discussed in this paper are powerful enough to avoid 15 out of 17 concurrency bugs that we analyzed.

4. ARCHITECTURAL SUPPORT

We implemented a profiler that learns PSets from the test runs using Pin [13]. We also implemented a runtime monitor to detect PSet constraint violations and avoid concurrency bugs using Pin [13]. The runtime overhead for this runtime monitor is about 100x for server applications such as MySQL and Apache, but it is over 200x for memory intensive applications like Splash [24] and Parsec [2]. Therefore, to constrain the interleavings at runtime using PSet constraints, adequate processor support is a must.

We discuss architectural support for detecting PSet constraint violations in this section. In addition, we also need checkpoint support for rollback and re-execution. This is a well researched problem. A copy-on-write mechanism can be supported in the operating system [20] or in the processor [22, 18]. During re-execution, we induce a different thread interleaving. The execution cannot be rolled back past a committed system state. But as we show in Section 5, the rollback window length required to avoid a majority of concurrency bugs is small.

We now discuss architectural support for detecting PSet constraint violations. The instruction set architecture (ISA) needs to be extended to let the developers specify the PSet constraints. Section 3 discussed an instruction encoding for specifying the PSet constraint for an instruction. A processor needs to execute a check for a memory operation, if it has a PSet constraint specified in the instruction code.

4.1 Tracking Last Writer and Last Reader(s)

To execute the checks, the processor needs to keep track of either the last writer or the set of last readers for every memory location. We propose to extend the caches to keep track of this additional meta-data for every memory location. When a cache block is evicted, the information is lost. But as described in Section 3, most of the concurrency bugs are tightly interleaved. Therefore, the loss in information due to a cache eviction is not significant.

The coherence reply messages (write-update replies and acknowledgments for invalidations) are piggy-backed with the meta-data corresponding to the cache block. The processor core receiving the reply, stores the received information in its private cache along with the information that the last reader or the writer information belongs to a different thread.

4.2 Checking PSet Constraints

We propose to use DISE [3] for efficiently executing the PSet check for every memory operation that has a PSet constraint. A check needs to be executed for a memory operation, only if the last reader(s)/writer to the memory location accessed by the current instruction belongs to a different thread. Thus, in the common case, no check needs to be executed for a memory operation. Also, for a majority of instructions, the PSet is a null-set (including all thread local accesses). We show that less than 5% of static memory operations have a PSet size greater than one, and therefore the check for a memory operation could be very efficient. If a check is executed for a memory operation, it checks if the last writer or the last set of readers is a member of the current memory operation's PSet.

5. RESULTS

We discuss several results in this section. First, we discuss the bug avoidance capability of an interleaving constrained shared-memory multi-processor that enforces the PSet constraints. We analyze its capability in detecting *and* avoiding 17 concurrency bugs (16 real bugs and 1 injected bug) in several multi-threaded applications such as *Mozilla*, *MySQL*, *Apache*, etc. We also analyze if these bugs can be detected by a happens-before based data race detector [8] and AVIO [11]. Second, we analyze the number of tests it takes to learn the PSet constraints adequately, and compare it with another test based AVIO bug detection tool [11]. Third, we discuss the number of PSet constraint violations in real executions (using input different from the ones used for training), and the overhead in resolving the PSet constraint violations using stalling and rollback mechanisms. Finally, we analyze the size of PSets and the memory space overhead to express PSet constraints in the binary and to keep track of them during production runs. These results are based on our PSet constraint tool implemented using Pin [13].

5.1 Bug Avoidance Capability

We analyze 17 bugs in the following multi-threaded programs: *Pbzip2*, *Aget*, *Pfscan*, *Apache*, *MySQL*, *Mozilla* and *OpenLDAP*. These bugs are listed in Table 1. Our PSet based detection tool was able to detect four real bugs (`Bug #1`, `Bug #2`, `Bug #4` and `Bug #5`) and one injected bug (`Bug #3`) during a real program execution (the PSet constraints for these programs were derived from the correct test runs, which are described in Section 5.2.1). For the rest of the bugs, we analyzed their extracted versions, as these bugs manifest only under a very specific interleaving that is very difficult to reproduce and analyze. The proposed PSet constraint based detection tool detected all the bugs, except the last two bugs listed in Table 1. One bug (`Bug #16`) is related to an incorrect interleaving between memory operations accessing different locations. The other one (`Bug #17`) is a deadlock bug. In order to detect this bug, the PSet constraint needs to be context sensitive. AVIO [11] can detect 6 atomicity violation bugs, but cannot detect one atomicity violation bug (`Bug #10`), which we discussed in Section 3. A happens-before data race detector can detect all the bugs, except five data race free bugs (`Bug #3`, `Bug #7`, `Bug #15`, `Bug #16` and `Bug #17`).

Our PSet based tool detected `Bug #3` and `Bug #15`, which

Bug #	Program	D.R.D	AVIO	PSET	Category	Description
1	Pbzip2	Yes	No	Yes	Other Concurrency Bugs	An order violation between the main thread and the consumer threads. The consumer threads are expected to terminate before the main thread release the mutex `fifo->mut`. However, when this order is not enforced, it will lead to a segmentation fault.
2	Aget	Yes	Yes	Yes	Atom. Vio.	A data race on the variable `bwritten`. The bug occurs when users issue `ctrl+c` from the console. Since the signal handler accesses the variable `bwritten` without holding the mutex lock, it is likely that the downloaded file and the log file are inconsistent.
3	Pfscan	No	No	Yes	Other Concurrency Bugs	An injected bug. A counter, which specifies the number of remaining threads, is used by the main thread to wait until all the child threads finish execution. The main thread should initialize the counter before any child thread finishes execution. Otherwise, the main thread will deadlock.
4	Apache	Yes	Yes	Yes	Atom. Vio.	A bug in Apache-2.0.48 that results in incorrect logs. Multiple threads call `ap_buffered_log_writer()` and change the buffer entry length simultaneously, corrupting the log file.
5	MySQL	Yes	Yes	Yes	Atom. Vio.	A security bug in MySQL-4.0.12. If a database update occurs when an old bin log is closed and the new bin log is not yet opened, this update will not be recorded, leading to a serious security problem.
6	MySQL	Yes	No	Yes	Other Concurrency Bugs	A bug usually occurs when MySQL starts. Due to unintentional interleaving, an uninitialized value is read by a thread, resulting in all data nodes becoming a master node.
7	Mozilla	No	Yes	Yes	Atom. Vio.	An atomicity violation bug in Mozilla [11]. While one thread loads a script and compiles it, the other thread nullify the script. This leads to a program crash. This bug is free of data race.
8	Mozilla	Yes	Yes	Yes	Atom. Vio.	An atomicity violation bug in `nsZipArchive.cpp`. Two different threads call `SeekToItem()` simultaneously, leading to one piece of code, which is supposed to be executed only once, get executed twice. One thread then will read garbage data.
9	Mozilla	Yes	Yes	Yes	Atom. Vio.	A bug in `nsNSSComponent.cpp`. While closing Mozilla, two threads check and free simultaneously. If two threads interleave incorrectly, a lock will get freed twice, thus cause a crash.
10	Mozilla	Yes	No	Yes	Atom. Vio.	A race between `nsFileTransport.cpp` and `nsAsyncStreamListener.cpp`. There is one temporary state which should be invisible to other threads. Due to bad interleaving, this temporary state can be read by another thread, leading to a deadlock.
11	Mozilla	Yes	No	Yes	Other Concurrency Bugs	An order violation in `nsthread.cpp` [10]. It is possible that a thread reads a location that is not initialized yet. This will cause Mozilla to crash.
12	Mozilla	Yes	No	Yes	Other Concurrency Bugs	A race between `macio.c` and `macthr.c` [10]. A callback function is expected to be invoked after a thread writes to a variable, but it is not synchronized properly. This bug will cause a deadlock.
13	Mozilla	Yes	No	Yes	Other Concurrency Bugs	An order violation in `nsImapProtocol.cpp`. An event will be ignored when certain interleaving occurs. This will cause Mozilla to hang.
14	Mozilla	Yes	No	Yes	Other Concurrency Bugs	An order violation in `nsHttpdConnection.cpp`. `OnHeadersAvailable()` is expected to be called after `AsyncWrite()` returns, but not enforced. It will crash Mozilla.
15	Mozilla	No	No	Yes	Other Concurrency Bugs	A bug in `TimerThread.cpp`. It happens when `Shutdown()` is executed prior to `Run()`. In that case, the exit event is missed and leads to a freezing state.
16	Mozilla	No	No	No	Multi-var. Atom. Vio.	An atomicity violation that involves multiple variables [10]. An inconsistent state is observed by a remote thread, causing corrupted memory.
17	OpenLDAP	No	No	No	Deadlock	A deadlock bug in `back-bdb/cache.c` (also reported in [23]). `bdb_cache_add()` is called by two threads. One thread holds the lock `lru_mutex` and try to acquire the lock `c_rwlock`, while the other thread holds the lock `c_rwlock` and try to acquire the lock `lru_mutex`, leading to a deadlock.

Table 1: Bug descriptions.

Bug #	Program	Type	Stall	Rollback	True Constraint Violations		False Constraint Violations		Rollback Window Size
					Static	Dynamic	Static	Dynamic	
1	Pbzip2	Real	Yes	Yes	1	1	3	3	0
2	Aget	Real	No	Yes	1	1	2	2	11
3	Pfscan	Injected	No	Yes	1	1	0	0	51
4	Apache	Real	No	Yes	2	20	1	1	358
5	MySQL	Real	Yes	Yes	1	7	3	6	0
6	MySQL	Extract	No	Yes	1	1	0	0	4760
7	Mozilla	Extract	No	Yes	1	1	3	3	1664
8	Mozilla	Extract	No	Yes	2	2	1	1	1224
9	Mozilla	Extract	No	Yes	1	1	0	0	1210
10	Mozilla	Extract	Yes	Yes	1	1	0	0	0
11	Mozilla	Extract	Yes	Yes	1	1	0	0	0
12	Mozilla	Extract	Yes	Yes	1	1	0	0	0
13	Mozilla	Extract	No	Yes	1	1	0	0	821
14	Mozilla	Extract	Yes	Yes	1	1	0	0	0
15	Mozilla	Extract	No	Yes	2	2	1	1	1674

Table 2: Avoiding bugs using PSet constraints. True constraint violations are related to the bug.

neither the data race detector nor AVIO [11] could detect. Thus, PSet constraint based concurrency bug detector is effective in detecting all the concurrency bugs that traditional tools find, and also has the potential to detect other memory ordering related concurrency bugs.

We now analyze the bug *avoidance* capability of the proposed constrained shared-memory multi-processor runtime system. Table 2 shows all the 15 bugs that were detected by the PSet violation detector. Six bugs were avoided using the stalling mechanism that we described in Section 3.4, and the rest of the bugs require support for a rollback and re-execution mechanism. Table 2 also lists the number of static and dynamic PSet constraint violations detected by our tool. The constraint violations are classified into true and false constraints. The true constraint violations are related to the bug. The false constraint violations are due to insufficient training during testing. The performance impact due to false constraint violations is discussed in Section 5.3. Table 2 also lists the number of instructions that need to be rolled back to avoid the bugs that we analyzed. As expected, the required rollback window size is small. This is because, most of the concurrency bugs are due to temporally tight interleaving between the memory operations. A rollback window of size zero means that the bug was avoided by just stalling the violating thread.

5.2 Learning PSet Constraints

For the runtime system to be efficient, PSet constraints should be complete enough to allow valid frequent interleavings between memory operations at runtime. In this section we discuss how soon the number of new PSets learned reaches a saturation point, and compare it with another profiling based bug detection tool called AVIO [11].

5.2.1 Testing Methodology

PSet constraints used in Section 5.1 were learned from the correct test runs. Here we describe the input we used to test our multi-threaded programs and learn the PSet constraints. These input are different from the ones used for the bug avoidance (Section 5.3) and the false positive analysis (Section 5.3). *Pbzip2* is a parallel implementation of *Bzip2*, which does file compression and file decompression. We compressed a random file in each test run. *Aget* is a download accelerator that spawns multiple threads to download different chunks of a file in parallel. For each test run, we downloaded a random file from the Internet. *Pfscan* is a multi-threaded file scanner, which combines the functionality of `find`, `xargs` and `fgrep`. We searched a random string from a randomly chosen file or directory in each test run. We also evaluated two server applications, *Apache* and *MySQL*. For *Apache*, each test run consists of issuing a session of requests to a set of static web pages using `httperf`. For *MySQL*, each test run consists of running the regression test suite that is available for public. In parallel with the regression test suite, we also continuously run the *OSDB* (Open Source Database Benchmark [1]) multi-user test to emulate a concurrent workload. For these five programs, we used the same version as the ones used for the bug avoidance analysis (from `Bug #1` to `Bug #5`), and the PSet constraints derived in this section are used in the bug avoidance analysis. In addition, we also evaluated six bug free applications, four of them (*FFT*, *LU*, *Radix*, *FMM*) are from the *Splash2* [24] benchmark suite, and two of them (*Blackscholes* and *Canneal*) are from the *Parsec* [2] benchmark suite. For these programs, we chose a random input parameter for each test run.

Programs	Stall	Rollback	Cannot Resolve	Total PSet Constraint Violations	Inst. Count
pbzip2	1	5	0	6	1.3E+9
aget	0	0	0	0	1.1E+7
pfscan	1	2	0	3	7.4E+7
apache	1	4	0	5	2.8E+8
mysql	0	2	2	4	9.7E+8
fft	0	0	0	0	2.3E+8
fmm	1	0	0	1	1.6E+9
lu	0	1	0	1	1.6E+8
radix	0	0	0	0	6.4E+7
blackscholes	0	0	0	0	8.1E+8
canneal	1	0	0	1	7.0E+9

Table 3: PSet constraint violations in bug-free executions.

5.2.2 Tests Required to Learn PSet Constraints

Figure 8 shows the number of new PSet pairs learned in each test run. Each point along the x-axis represents a unique test run, and the y-axis represents the number of new PSet pairs derived from a particular test run. PSet takes more test runs to stabilize than AVIO, because it captures more constraints than AVIO. These results show that the tests used during the quality assurance process should be adequate to learn the PSets.

5.3 PSet Constraint Violations in Bug Free Executions

We now discuss the number of false PSet constraint violations in bug free executions. We used the same set of benchmarks that we used for the results in Section 5.2.2. The input used to analyze the false PSet constraint violations is different from the training input. For *Pbzip2*, we used a different set of files as input. For *Aget*, some new files were downloaded. For *Pfscan*, we searched some new strings from different files and directories. For *Apache*, we used `httperf` to issue concurrent requests to a set of static web pages which are not used in the test runs. For *MySQL*, we used the tool in *OSDB* to randomly generate a new database with a size different from the one used for training, and ran the *OSDB* multi-user test. For *Splash2* and *Parsec* programs, we randomly selected inputs and parameters not used in the test runs.

Table 3 shows the static and dynamic number of PSet constraint violations, and also the total number of instructions executed. All the data shown in the table are the cumulative results of 10 runs, except for *MySQL*. For *MySQL*, we run the *OSDB* multi-user benchmark once. The results show that the number of false constraint violations are very few. For example, for *Pbzip2*, we detected 6 constraint violations while executing over 1.3 billion instructions. We avoided one constraint violation by just stalling the violating thread. The other five constraint violations needed rollback and re-execution to resolve them. Even for a rollback window of length 100,000 (which is more than sufficient for resolving most of the concurrency bugs as discussed in Section 5.1), five violations would result in additional 0.5 million instructions being executed at runtime. But this is a small fraction (0.04%) when compared to 1.3 billion instructions executed

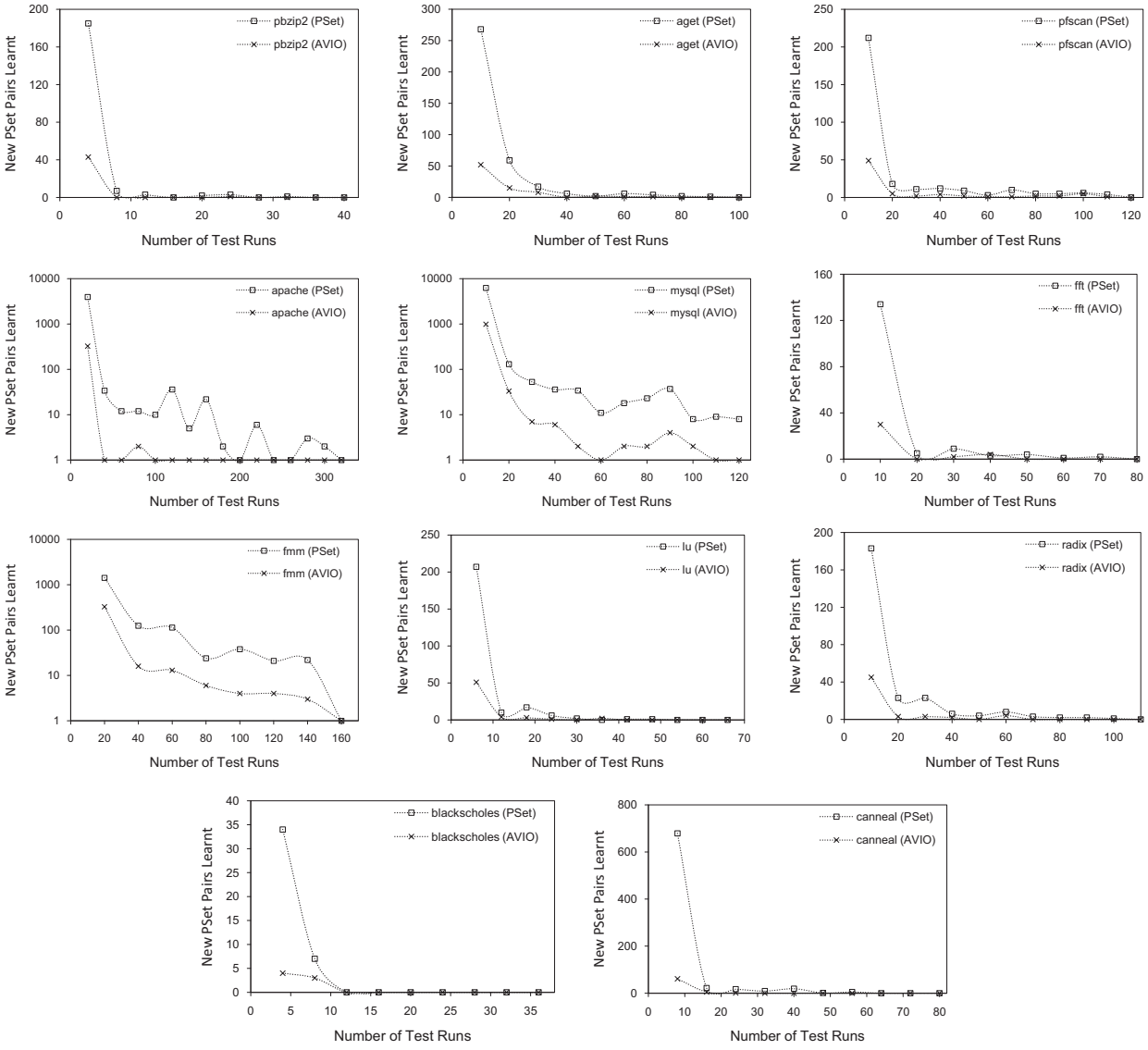

Figure 8: Number of test runs required for learning PSets and AVIO invariants.

Programs	App. Size	App+Lib Size	# PSet Pairs	PSet Size	Overhead w.r.t App.	Overhead w.r.t App+Lib
pbzip2	39KB	3.70MB	201	0.84KB	2.16%	0.02%
aget	90KB	2.04MB	365	1.53KB	1.69%	0.08%
pfscan	17KB	2.08MB	295	1.25KB	7.34%	0.06%
apache	2435KB	8.60MB	4119	16.80KB	0.69%	0.20%
mysql	4284KB	8.19MB	6604	27.58KB	0.64%	0.34%
fft	24KB	2.59MB	158	0.67KB	2.74%	0.03%
fmm	73KB	2.64MB	1764	7.39KB	10.13%	0.28%
lu	24KB	2.59MB	244	1.03KB	4.31%	0.04%
radix	21KB	2.59MB	255	1.07KB	5.00%	0.04%
blackscholes	54KB	3.65MB	41	0.17KB	0.32%	0.00%
canneal	59KB	3.66MB	752	3.10KB	5.24%	0.08%

Table 4: Binary Size Increase

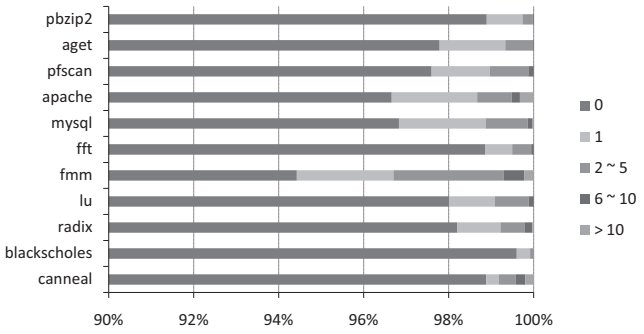

Figure 9: Proportion of static memory instructions with a particular PSet size (normalized to the total number of static memory instructions in the application binary and libraries that were executed in at least one test run).

by the original program. Thus, false constraint violations are infrequent enough that they are likely to not impact performance.

Detecting PSet invariant violations and maintaining checkpoints can also degrade performance, but both of these costs can be ameliorated using processor support (discussed in Section 4).

We also noticed that there are two PSet constraint violations in *MySQL* that cannot be resolved by both stall and rollback mechanisms, because there is no legal interleaving that would not violate the PSet constraints. This is due to insufficient testing. We found that these two violations are from the function `bmove512()` (in `bmove512.c`). This function moves 512 bytes in the heap. To improve performance, the programmer has manually unrolled a loop in the function 128 times resulting in 128 reads and 128 writes within the loop. In a test run, only very few bytes are touched in the heap before the function `bmove512()` gets executed. As a result, in a test run, only a few memory instructions within the loop have a predecessor memory operation. This makes it difficult for us to learn all the legal interleavings involving this function using our current testing methodology. However, if we can perform more complete industry-level testing, such violations should also disappear.

When we detect a PSet violation that we cannot avoid, in addition to logging it for post-mortem analysis, the corresponding PSet is updated so that no future violations due to the same interleaving would be detected. A PSet violation that we manage to avoid are also logged for post-mortem analysis, but the corresponding PSet is *not* updated so that the system continues to avoid a potential bug related to the PSet violation.

5.4 Memory Space Overhead

We now discuss the memory space overhead in expressing the PSet constraints in the binary and tracking them at runtime. We use the same set of benchmarks that we described in Section 5.2.2.

Figure 9 shows the distribution of the PSet sizes for several programs that we analyzed. The x-axis is normalized to the total number of memory instructions in the application binary and the libraries that were executed at least once in the test runs. Over 95% of the instructions have PSets of size zero. These instructions either accessed only thread local memory locations, or, they were never dependent on a remote memory operation. Less than 2% of static memory instructions have a PSet of size greater than two. This result shows that the performance overhead in the executing the PSet constraint checks should be very small.

Table 4 lists the sizes of the applications' binaries and also the sizes of the libraries they use. It also lists the number of PSet pairs learned from the test runs. The percentage of code size increase with respect to just the application binary size is about 10% in the worst case. This is the code size increase for an application. However, the PSet pairs for an application also includes the static instructions from the libraries. The increase in binary size with respect to the total size of application and libraries is negligible. As a result, at runtime, we expect that the increase in the size of the instruction memory footprint to be also negligible. The reason for this result is that, only a small fraction of the instructions access shared-memory locations. And, only the shared-memory instructions could have a PSet of size greater than zero.

6. CONCLUSION

Testing and verifying a multi-threaded program is more difficult than a single-threaded program, because the number of possible interleavings is exponential over the number of memory operations executed by different threads. We make a case for an interleaving constrained shared-memory multi-processor which avoids untested interleavings.

This paper makes the first step towards designing an interleaving constrained multi-processor. To detect untested interleavings we need a set of invariants fundamentally different from the ones used to detect incorrect interleavings such as a data race invariant or AVIO. We focused on constraining the runtime interleaving such that, no two remote memory operations are allowed to depend on each other at runtime, unless that dependency was observed in at least one of the test runs. We built a software tool to detect PSet constraints and enforce them, but as expected, it incurs significant runtime slowdown. We proposed extensions to a multi-processor design, which enables efficient detection of PSet constraint violations. On detecting a violation, checkpoint support is used for re-executing the program with an alternative interleaving and resolve the PSet constraint violations.

We analyzed several bugs in real applications, and showed that the proposed system can avoid not only data races and atomicity violations, but also other unstructured memory order related concurrency bugs. The number of false constraint violations in a well tested program is very small, and as a result, the resulting performance overhead is also negligible.

Acknowledgments

We would like to thank the anonymous reviewers for providing valuable feedback on this paper.

7. REFERENCES

[1] The open source database benchmark. http://osdb.sourceforge.net/.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th*

International Conference on Parallel Architectures and Compilation Techniques, October 2008.

[3] M. Corliss, E. Lewis, and A. Roth. Dise: A programmable macro engine for customizing applications. In *30th Annual International Symposium on Computer Architecture*, San Diego, CA, June 2003.

[4] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: deterministic shared memory multiprocessing. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 85–96, New York, NY, USA, 2009. ACM.

[5] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 256–267, 2004.

[6] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency (tcc). In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 1–13, New York, NY, USA, 2004. ACM Press.

[7] M. Herlihy, V. Luchangco, M. Moir, and W.N.Scherer. Software transactional memory for dynamic-sized data structures. In *Twenty-Second ACM SIGACT-SIGOPS Symposium on Princicples of Distributed Computing*, 2003.

[8] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[9] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. Muvi: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, 2007.

[10] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes — a comprehensive study on real world concurrency bug characteristics. In *13th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'08)*, 2008.

[11] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 37–48, 2006.

[12] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-aid: Detecting and surviving atomicity violations. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, pages 277–288, Washington, DC, USA, 2008. IEEE Computer Society.

[13] 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.

[14] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI)*, June 2007.

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

[16] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 97–108, New York, NY, USA, 2009. ACM.

[17] 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.

[18] 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.

[19] S. Qadeer. Taming concurrency: A program verification perspective. In *Invited Lecture. International Conference on Concurrency Theory (CONCUR)*, Aug 2008.

[20] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 235–248, New York, NY, USA, 2005. ACM.

[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] 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, 2002.

[23] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. A. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *OSDI '08: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, December 2008.

[24] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *22nd Annual International Symposium on Computer Architecture*, pages 24–36, 1995.

[25] M. Xu, R. Bodik, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, 2005.