PASSERT: A Tool for Debugging Parallel Programs

Daniel Schwartz-Narbonne*, Feng Liu, David August, and Sharad Malik*

Princeton University
{dstwo,fengliu,august,sharad}@princeton.edu

Abstract. PASSERT is a new debugging tool for parallel programs which allows programmers to express correctness criteria using a simple, expressive assertion language. We demonstrate how these *parallel assertions* allow the detection and diagnosis of real world concurrency bugs, detecting 14/17 bugs in an independently selected set of bugs from open source software. We describe a runtime checker which allows automatic checking of parallel assertions in C and C++ programs, with a geometric mean of 6.6× overhead on a set of PARSEC benchmarks. We improve performance by introducing a *relaxed timing semantics* for parallel assertions, which better reflects real memory models, and exposes more bugs with less overhead (geometric mean overhead 3.5×).

1 Introduction

PASSERT is a new debugging tool for parallel programs which allows programmers to express correctness criteria using a simple, expressive assertion language. If a correctness property is violated during program execution, an automatic runtime checker will detect the violation.

Such a tool is necessary because the standard assertions that are widely used for debugging sequential programs are highly limited for parallel programming. In a sequential program, it is sufficient to check whether a property holds at a particular point in time. In a parallel program it is possible for a property to be true when a section of code begins executing, for the code in question to make no changes that could falsify the property, and yet for the property to be violated by the actions of a second thread. Checking whether a property holds during execution through a small code segment potentially requires annotating every statement of the program with assertions. Even this might not be sufficient: since assertions are not synchronized with code execution, checking an assertion that depends on more than one program variable might be impossible to do correctly without significant code modification.

Parallel assertions, the input language of PASSERT, solve this problem by providing a simple, understandable set of predicates that allow programmers to write local assertions which allow testing of multithreaded programs.

* The authors acknowledge the support of the Gigascale Systems Research Center (GSRC), one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation (SRC) entity.

2 The Parallel Assertion Language

2.1 Syntax

A programmer debugging a piece of code wants to know whether a property holds during the execution of a piece of code. A parallel assertion therefore consists of two parts: a description of the times when the assertion is expected to be true, which we call the *assertion scope*, and the property which is expected to hold, which we call the *assertion condition.* Parallel assertions are expressed using simple, easy to use predicates. More powerful formulations, such as temporal logic, are possible but are unfamiliar to programmers and provide expressiveness at the cost of complexity.

Parallel assertions are applicable to many programming languages. This paper focuses on our C/C++ implementation; a full formal syntax and semantics is provided in [3].

Assertion Scope. The assertion scope is a block of code which delineates the time during which the assertion condition must hold. It begins with the keyword `thru` and ends with `passert(cond)`.

```
thru {
  ... ;
} passert (cond)
```

Assertion Condition. The assertion condition is a side-effect free Boolean expression, which can contain Boolean combinations of any of the following sub-expressions:

Type	**Description**	**Example**
Value	Any side effect free boolean expression	$x > 5$
LocalRead(x)	True when the asserting thread is reading the variable x	LR(x)
LocalWrite(x)	True when the asserting thread is writing the variable x	LW(x)
RemoteRead(x)	True when a thread other than the asserting thread is reading the variable x	RR(x)
RemoteWrite(x)	True when a thread other than the asserting thread is writing the variable x	RW(x)
HasOccurred(expr)	True iff expr has ever been true while the assertion was active	HO(expr)

2.2 Assertion Semantics

The result of a parallel assertion is defined relative to a program execution i.e. an observed interleaving of read and write events by the executing threads. We augment this program execution by adding *assertion begin* and *assertion end* events, which mark the beginning and end of assertion scopes. A *timeline* is an observed total ordering of these events for a particular execution of a parallel program. (Under certain circumstances, discussed in Sec. 6.2, this requirement for a total ordering can be relaxed).

An assertion holds, for a particular timeline, if the assertion condition is true for all times between the beginning and end assertion events. It fails if there is any time between the begin and end events during which the assertion condition is not true. Since assertions are checked, not enforced, they can be used for debugging and then turned off for production.

3 Applicability of Parallel Assertions to Real World Bugs

Assertions have two purposes: to detect unexpected events that may represent bugs, and to test hypotheses to diagnose the cause of these bugs. We evaluated the effectiveness of parallel assertions using the University of Michigan Collection of Concurrency Bugs [4][1], an independently selected collection of real world concurrency bugs from major open source programs. For each bug, we attempted to write a parallel assertion to detect its symptoms and diagnose its underlying cause, without requiring any other code modifications.

Bug Analysis Example MySQL-3.23.56 had a concurrency bug which caused it to produce a nonsensical log: for example, it could report a successful insert before the associated table had been created. We identified several possible explanations for this bug, and wrote a parallel assertion to test each of these.

– Assertion 1 checked for a data-race problem and confirmed that all accesses to the log are protected by a lock.
– Assertion 2 determined that inserts never occurred while the table was invalid.
– Assertion 3 tested whether the log order represents the actual order of events, i.e. does an operation (such as creating a table), and the logging of that operation, form a single atomic unit. This seemingly simple test requires the expressiveness of parallel assertions. It would be incorrect to mark the entire `generate_table()` function as atomic, because it correctly accesses shared variables in a non-transactional way. In addition, while conflicting writes to the logger represent an error, reads may not. This assertion captures these subtleties, and successfully diagnosed the cause of the error.

These assertions could subsequently be left in the program as regression tests.

```
int generate_table(...) {
  ...
  thru{
    pthread_mutex_lock(&LOCK_open);
    // delete the original table
    // create a new table
    pthread_mutex_unlock(&LOCK_open);
    mysql_update_log.write(...);
  }passert(!RW(mysql_update_log));
  ...
}
```

```
bool MYSQL_LOG::write(...) {
  ...
  pthread_mutex_lock(&LOCK_log);
  // log event
  pthread_mutex_unlock(&LOCK_log);
  ...
}
```

(Parallel assertion to identify MySql bug 169)

[1] Currently maintained at `http://www.eecs.umich.edu/~jieyu/bugs.html`

Summary of Bug Coverage Almost all (14/17) of the bugs in the University of Michigan Collection can be detected using parallel assertions (shown in the table below). Of these, thirteen can also be diagnosed using parallel assertions; one is a multi-function atomicity violation, which would be difficult to capture in a single syntactic scope. PASSERT is not designed to detect deadlock and complex order-violation bugs.

Bug ID	**Bug Type**	**Detect Symptom**	**Diagnose Cause**
Apache #25520, 21287 MySQL #44, 791, 2011, 3596, 12848 Cherokee Bug1, Aget Bug2	Data Race	Yes	Yes
Apache #45605 MySQL #169, 12228 Memcached #127	Atomicity	Yes	Yes
Apache #21285	Atomicity	Yes	No
Pbzip2 0.9.4, Transmission 1.42	Order Violation	No	No
Aget Bug1	Deadlock	No	No

4 Tool Design

PASSERT is a compiler that automatically adds runtime support for parallel assertions in C/C++ programs. It is implemented as an extension to the LLVM [2] compiler suite, and supports the same programs and language features as the standard LLVM compiler. At present, PASSERT targets programs using the *pthreads* threading library; we expect that it will be easy to extend it to other threading models, such as Windows threads.

We reduced the impact of assertion evaluation on program execution by decoupling execution and checking: as the program executes, relevant loads/stores are timestamped and logged for subsequent checking.

Logging. To reduce logging overhead, we use alias analysis to determine whether a load or store may access a variable in a parallel assertion condition. Since static alias analysis is overly conservative, PASSERT also maintains a hash-table which records whether accesses to a given memory location need to be logged. Collisions in the hash-table may cause unnecessary logging, but will never cause an event to be missed from the log.

Checking. Checking can either be done online, using a separate checking thread and synchronized queues, or offline, in which case no synchronization needs to be done on the queues. Performance results are discussed in Sec. 7.

Avoiding Stalls. An event can only be processed if all events which occur before it in the execution trace have already been processed. If a thread stops generating events, it becomes impossible to determine the correct sequence of events, since the checker has no way of distinguishing between "no event" and "a not-yet-logged event". The checker must therefore conservatively wait until the thread resumes generating events. If a thread which is about to stall can generate a *Thread Stalled* event, the checker can continue without waiting. PASSERT automatically generates such events before calls to blocking functions such as `pthread_join()`, `pthread_barrier_wait()` and `pthread_cond_wait()`. Programmers writing specialized synchronization libraries can add their own event annotations. They can also insert *Heartbeat Events* into code which is unlikely to generate any logged events, such as calculations on privatized data. As a future extension, we hope to introduce heuristics that will automatically add these events at appropriate points.

5 Response to Assertion Failure

When an assertion fails, PASSERT informs the user and prints out a set of diagnostic information. This information includes which memory access caused the assertion failure (including time, thread_id, value, and type of access), as well as which assertion was triggered. If the user desires, PASSERT can output its full log to a file. If the program has been compiled with debug symbols, it is possible to associate the log information with program locations, although this is not currently implemented. A compiler flag controls whether the executable should abort or continue after an assertion violation.

In addition, PASSERT provides a feedback function which allows user code to block until the checker has evaluated all events before the feedback function call began, and then returns the checker status (i.e. failure or success). The program can use this mechanism to ensure that a dangerous action only occurs after correct execution, or to rollback and recover after an assertion failure. As a convenience, PASSERT can automatically insert a checker feedback call at the end of each `thru` block.

6 Timing modes

The semantics of parallel assertions, as introduced in [3], requires a total ordering of events during a concurrent execution. However, modern microprocessors typically have more relaxed semantics, which allow for event sequences which do not have any consistent total order.

6.1 Strict Timing

In *strict timing mode*, this total ordering is enforced through the use of locks and fences around every logged memory access. Timestamps can be acquired either through a global counter, or through a hardware timestamping mechanism such as RDTSC [1].

6.2 Relaxed Timing

Not all parallel assertions require a total order over program events in order to be correctly evaluated. In some cases, a partial order among certain types of events is sufficient. In particular, any assertion which either:

– Only contains access predicates (such as RemoteWrite(x)), or
– Contains value predicates, but only references a single variable

requires a partial order between access and assertion begin/end events, but does not require any further ordering among access events.

Relaxed Timing Mode enforces only these minimal constraints, dramatically reducing the number of locks and memory fences required. This both reduces runtime overhead (see Sec. 7), and allows a wider range of bugs to manifest, since locks prevent certain combinations and orderings of events that would be legal and possible in the underlying hardware model.

7 Results

We evaluated the runtime performance of PASSERT using the assertion-annotated PARSEC benchmarks described in [3]. All benchmarks were compiled at optimization level O3, and were executed on a quad core Intel X3440 with 16GB of RAM. The runtime for each benchmark, normalized to the unmodified benchmark compiled using standard gcc, is reported in Fig. 1.

The online checker performs checking in parallel with execution, which speeds up the checking phase, but requires extra synchronization in the logging phase. Currently, these two effects roughly cancel each other out; we hope to remove this overhead with further optimization. Strict timing had a geometric mean overhead of 6.6×. Relaxed timing was significantly faster, with a geometric mean overhead of 3.5×.

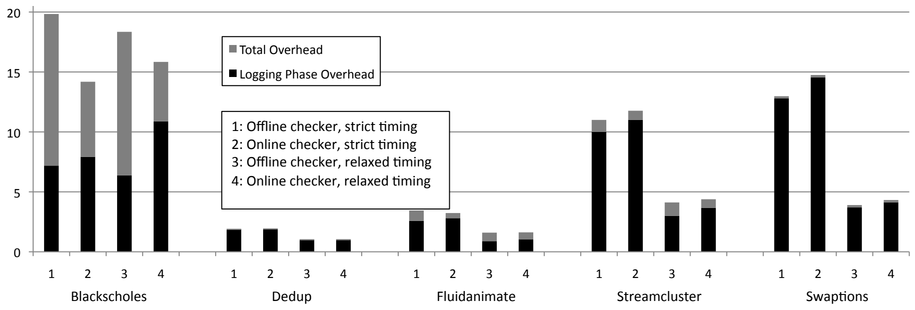

Fig. 1. Runtime overhead for parsec benchmarks

8 Conclusion

PASSERT provides programmers with a new tool to identify elusive bugs in parallel programs. Until now, parallel programs have been challenging to debug, because it has been hard to express and check assumptions about program execution across multiple threads. Our experience with the Michigan Bug Collection shows that parallel assertions are sufficiently expressive to capture a range of real-world bugs. Our performance experiments indicate that checking these assertions can be done with reasonable overhead. The simple, expressive syntax of PASSERT allows programmers to express correctness conditions to debug programs with a high degree of efficacy and a minimum of effort.

References

1. Intel Corporation. Intel 64 and IA-32 Architectures Developer's Manual (2010)
2. Lattner, C.: LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL (2002)
3. Schwartz-Narbonne, D., Liu, F., Pondicherry, T., August, D., Malik, S.: Parallel assertions for debugging parallel programs. In: MEMOCODE 2011, pp. 181–190 (2011)
4. Yu, J., Narayanasamy, S.: A case for an interleaving constrained shared-memory multi-processor. In: ISCA 2009, pp. 325–336 (2009)