Protecting Synchronization Mechanisms of Parallel Big Data Kernels via Logging

Travis LeCompte[1], Lu Peng[1], Xu Yuan[2], and Nian-Feng Tzeng[2]
[1]Louisiana State University, [2]University of Louisiana at Lafayette

***Abstract*—With the growing effort to reduce power consumption in machines, fault tolerance becomes more of a concern. This holds particularly for large-scale computing, where execution failures due to soft faults waste excessive time and resources. These large-scale applications are normally parallel in nature and rely on control structures tailored specifically for parallel computing, such as locks and barriers. While there are many studies on resilient software, to our knowledge none of them focus on protecting these parallel control structures. In this work, we present a method of ensuring the correct operation of both locks and barriers in parallel applications. Our method tracks the memory locations used within parallel sections and detects a violation of the control structures. Upon detecting any violation, the violating thread is rolled back to the beginning of the structure and reattempts it, similar to rollback mechanisms in transactional memory systems. We test the method on representative samples of the BigDataBench kernels and find it exhibits a mean error reduction of 93.6% for basic mutex locks and barriers with a mean 6.55% execution time overhead at 64 threads. Additionally, we provide a comparison to transactional memory methods and demonstrate up to a mean 57.5% execution time overhead reduction.**

***Index Terms*—algorithmic resilience, barriers, fault tolerance, locks, parallel programming**

I. Introduction

Program reliability is a major concern throughout many fields of computing. Applications that cannot reliably produce correct solutions are hardly useful. As chip designers drive to reduce power consumption, the voltage levels separating logic 0 and logic 1 become closer and provide less of an error margin. This increases the probability of bit flips during execution, where a 0 becomes a 1 or vice versa. Depending on the locations of these bit flips, they can directly interfere with program behavior and produce unexpected results.

Errors during execution can reveal themselves in different ways, including hangs, crashes and silent data corruptions (SDC). Crashes are the most obvious, where the process simply exits suddenly. Hangs can be more deceptive as the application may still seem to be doing work while actually making no progress. Most difficult to detect during execution are SDC, where a value used by the program is modified without causing a crash or hang. For example, one of the operands for an addition is corrupted, resulting in an incorrect output value. This can become particularly dangerous when errors propagate from one variable to another as the program continues to execute [16]. These errors waste time and resources as they go undetected.

While hardware solutions like error correcting memory (ECM) exist [6], they can be expensive to apply and carry overhead. This has led to an interest in software methods for fault tolerance. These methods typically achieve good error coverage with varying overhead costs depending on their implementation details. Software fault tolerance methods [11] for computing involve both a detection stage and a correction stage respectively for identifying and recovering from errors. In order to trigger correction mechanisms, the detection mechanism must first identify an error. Most methods exploit features specific to the algorithms in question to identify these irregularities during execution [12], [16], or require replication and comparison of the process periodically during execution to ensure correct behavior [3]. Newer approaches relying on machine learning to identify program deviation have also been introduced [14].

Correction mechanisms typically utilize a form of checkpointing for error recovery [9], [13]. Checkpointing involves taking snapshots of the process during execution and restoring to a previous correct snapshot upon detecting an error [13]. This can be done at varying granularity and frequency based on the application. Checkpointing is a relatively simple method that works well for crashes and hangs, but can be insufficient for SDC as the error can go unnoticed and result in erroneous checkpoints. In order to alleviate this, some systems require saving multiple checkpoints and more frequent checkpointing, involving undesirably higher overhead.

These detection and correction methods are extremely important for large-scale computing, where processes may run for many hours or even days on multiple nodes. If an SDC occurs early during execution, the algorithm could run for a long duration before the error is noticed, wasting substantial time, resources and energy. These programs are typically parallel in nature, employing fundamentally different techniques to solve problems. They commonly rely in part on synchronization mechanisms such as locks and barriers for sharing information among threads and ensuring coherence. However, these mechanisms themselves can be vulnerable to errors, leading to error behavior that occurs only within parallel programs. To our knowledge, there is little previous work aimed at protecting these synchronization mechanisms from transient faults. Application checkpointing systems can solve crashes and hangs resulting from errors in synchronization mechanisms but require additional detection for SDC. Transactional memory has been proposed as a method to protect code executions from concurrency bugs [15], though its focus is on programmer errors not transient faults.

In this work we present a method for identifying and correcting violations of these synchronization mechanisms caused by transient faults via local logging systems. Tracking thread locations during execution reveals violations of the synchronization mechanisms. We implement a local checkpointing and recovery mechanism for the threads through Intel Pin [7] by exploiting the conceptual properties of these mechanisms. We include an investigation into the results of faults within these synchronization components to demonstrate the effectiveness of such methods, and a measurement of the overhead costs for implementation. Finally, we provide a comparison with transactional memory, another form of local logging and rollback for parallel systems that can act as an alternative for lock-based mechanisms. Our system implements similar logging mechanisms to an eager transactional memory system, but it benefits from simplified conflict detection when fewer conflicts can occur. Note that our mechanism also differs from conventional checkpointing in that it conducts logging at each parallel control structure (locks and barriers) in preparation for rolling back, if required, as opposed to collecting system execution states periodically or adaptively [13] under conventional checkpointing.

The contributions of this work can be summarized as follows:

- We examine vulnerability of BigDataBench kernels [1] to soft faults within concurrency control mechanisms during execution.
- We design and develop a logging mechanism based on transactional memory to detect and correct the resulting concurrency bugs by enforcing the control mechanisms.
- We demonstrate a mean 93.6% error coverage from the resulting concurrency bugs caused by these soft faults with a mean 6.55% overhead in the execution time at 64 threads.
- We compare the overhead of our developed logging mechanism against a full transactional memory system and find up to a mean 57.5% reduction in execution time overhead relative to transactional memory.

II. Background

A. Concurrency Control

As previously mentioned, many fault tolerant methods exploit program features to increase coverage and reduce overhead. We focus specifically on locks and barriers as our synchronization structures. These fundamental mechanisms provide building blocks for more complex parallel data structures. However, these locks and barriers perform different functions and present different vulnerabilities. Locks protect critical sections of code, where only one thread should enter at any given time. Violations can cause race conditions where multiple threads access values at the same time. Failing to unlock locks, or poorly coordinating the order with which a thread claims multiple locks, can also lead to deadlocks, halting execution progress. Locks can be employed in either a fine-grained or coarse-grained manner. Coarse-grained locking protects a large amount of code that may not all be needed by the thread. It is easier to implement at the cost of performance, as more threads compete for the critical section. Fine-grained locking by comparison enables more parallelism by locking small sections of code that are specifically necessary for the thread, but it is difficult to implement and may be more prone to deadlocking.

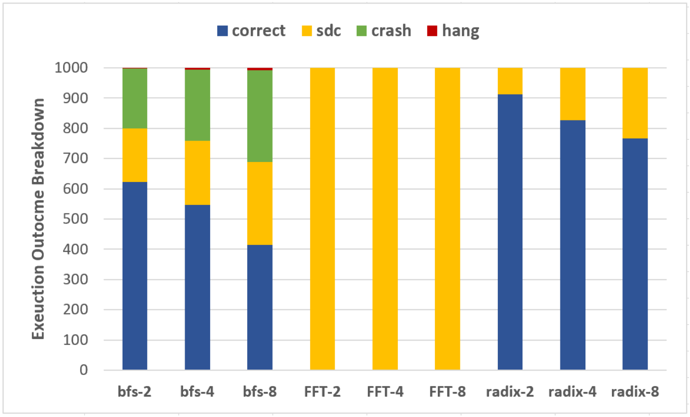

Fig. 1. Execution outcome breakdowns for various benchmarks under 2, 4, and 8 usable threads.

Barriers by comparison act as a trap, where no thread is allowed to pass until all involved threads have reached the barrier. Barriers are commonly used with an alternating computation and communication paradigm. When a thread finishes computation and needs to share information, each thread waits until every thread has completed computation and is ready for sharing. This prevents threads from overwriting values that are still needed by other threads, or reading old values that are no longer valid. Violations of the barrier would cause threads to sneak past, potentially causing these problems. Both locks and barriers are typically implemented using atomic operations that allow a thread to perform a combination of reads and writes as one single operation, ensuring coherence among multiple threads operating on a shared value. The most common example is the compare-and-swap (CAS), which compares a value m in memory to a given value v, and writes a third value to memory if m and v are equal.

B. Transactional Memory

There are other methods to ensure thread coherence besides directly using locks and barriers. The most relevant to note here is transactional memory. By automatically fine-grained locking individual memory locations, developers do not need to manually implement locking mechanisms. Instead, a thread simply marks the beginning and the end of a transaction, wherein all operations will be executed as if they were atomic. If there are conflicts due to multiple threads modifying the same memory locations, one thread is chosen to commit its transaction while others are forced to reattempt. These

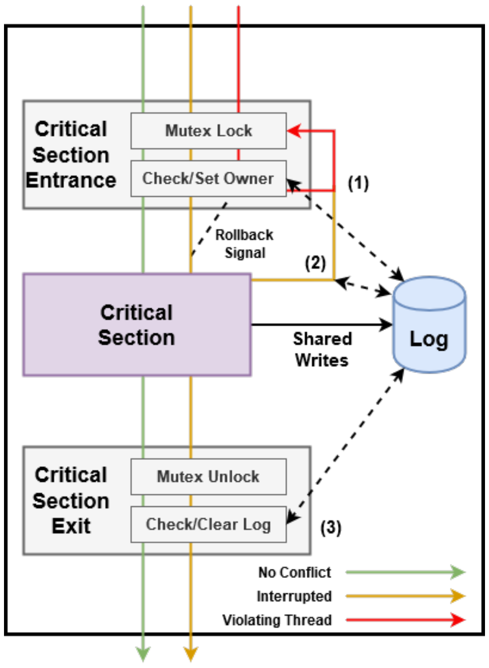

Fig. 2. Execution flowchart showing executions with no conflicts (green), immediate rollback (red) and forced rollback (orange).

transactional memory-based systems provide an alternative for developers, potentially allowing for greater parallelism in their fine-grained nature. By being aware of how threads operate on a memory location (read vs write), such a system can also allow multiple reads simultaneously as memory is then not modified. Transactional memory has previously been adopted to address concurrency bugs which result from developer errors but are not transient faults [15].

Transactional memory systems have been implemented both in hardware [10] and in software [4]. Both implementation methods have their respective benefits, with hardware systems typically having better performance in exchange for flexibility and simplicity. Transactional memory systems have also been proposed for accelerators like GPUs [2], [5]. As shown in the following sections, we utilize methods similar to eager transactional memory to protect coarse-grained locks and barriers. While similar to transactional memory, it is considerably simpler due to a limitation in the types of conflicts that may arise.

III. Motivation

It is important to note that transient errors in these parallel programs may differ considerably from those found in sequential programs. In sequential programs, faults may cause crashes, hangs or incorrect output by modifying pointers, loop control structures or variables holding important data. In parallel programs, crashes, hangs and SDC can all result from faults targeting parallel control structures like locks and barriers. For example, a fault that occurs in data used within or leading up to ”xchg” or CAS instructions may cause the synchronization mechanism to fail. These failures can result in crashes, hangs, or SDC when threads violate concurrency control, either through race conditions in critical sections or accessing improperly synchronized data. SDC caused by these failures can further propagate into different errors, which may be difficult to identify, locate and recover from. Due to the different nature of these errors, we make the first attempt to detect and correct them in non-traditional manners, as discussed in Section IV.

We aim demonstrate the importance of protecting concurrency control mechanisms in parallel applications by examining the vulnerability of three representative benchmarks of the BigDataBench kernels under 2, 4, and 8 threads. The full list representative benchmarks for the BigDataBench kernels is shown in Table I. We simulate transient faults using the Intel Pin Fault Injector (PINFI) [8]. As the errors can occur in different locations in each trial, we simulate transient faults rather than hardware faults. This automated fault injection tool targets instructions within select functions and modifies the bits to simulate soft errors during execution. For these experiments, we limit injection to the synchronization mechanisms contained within the programs. We inject a single fault into every lock encountered during execution. Hence, the number of faults injected for one trial equals the number of locks encountered over the course of execution. The results from these trials are shown in Fig. 1.

It is evident that as the number of threads increases, error frequency also increases. Having more threads in contention for the control structures results in more CPU time spent waiting at these structures during execution. As more instructions are executed involving these wait loops, it becomes more likely for errors to break these loops and thus break the control structures. It is worth noting the variety of error profiles among benchmarks. FFT is completely vulnerable to SDC, while radix is only mildly vulnerable. In comparison, BFS contains vulnerabilities not present in the other two, including a sizable number of crashes and some hanging executions. These outcomes are caused by soft fault injection into the control structures which can cause race conditions, deadlock, or direct crashes as shown.

Which outcomes are observed depends heavily on the algorithm itself. Programs like BFS that rely heavily on pointers experience more crashes as errors due to failed synchronization may corrupt these pointers and result in erroneous memory access. Conversely, programs that include many logical or arithmetic instructions like FFT are more vulnerable to SDC as errors due to failed synchronization are more likely to simply modify data and not cause crashes. This observation supports other works in that the use of algorithm-specific methods for detection and correction throughout the entire execution, may be more efficient than generic methods, confirming the importance of protecting locks and barriers.

IV. Design

Algorithm 1 Lock Error Detection and Correction

```
Input: The instruction pointer $p$ for the entry point of the
    critical section, the current thread id $tid$, and a dictionary
    of lock ownership status $S$
1: Atomically attempt to set $S[p] = tid$
2: if $S[p] = tid$ then
3:     Current thread successfully marked as owner, proceed
       to the critical section
4: else
5:     Send rollback signal to $S[p]$
6:     Clear $S[p]$
7:     Roll back to $p$ to reattempt lock acquisition
8: end if
```

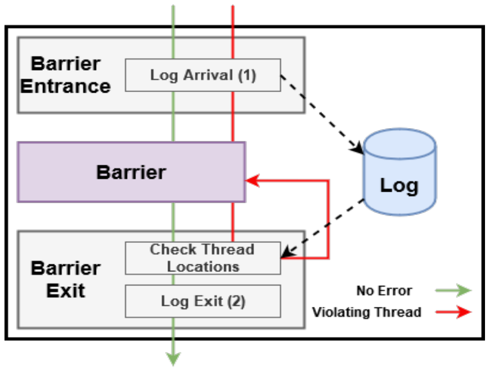

Fig. 3. Execution flowchart of barrier protection showing correct (green) and erroneous (red) threads.

A. Detection

In order to detect these violations, we implement a logging mechanism through Intel Pin [7] similar to an eager transactional memory system. The tool identifies marked barriers and locks in the binary and tracks their program locations during execution. This allows us to identify at what points which threads reside in critical sections or beyond barriers. Whenever multiple threads are detected within a critical section simultaneously, we know that its associated lock has been broken by faults. Similarly, if a thread ever passes a barrier before other threads are able to reach it, we know that the barrier had broken. Fig. 2 provides an example of both successful and recovered execution paths.

Before entering a critical section, a thread must pass through both the original lock and the following protection functions, marked as **(1)** in Fig. 2. The log tracks a thread's entrance to the critical section and executes Algorithm 1 to detect if the thread violates the exclusivity of the critical section. The green line shows a thread which executes without interruption. The orange line (thread 1) shows a thread that is interrupted whereas the red thread (thread 2) erroneously breaks the lock and enters the critical section. When thread 1 enters the critical section first, it is marked as the owner of the lock within the log and can progress into the critical section. When thread 2 enters the critical section before thread 1 has exited and released ownership, the logging system is aware that the lock has broken for thread 2 or thread 1 and thus correction is attempted.

Additional work is necessary to ensure locks are correctly unlocked when leaving a critical section. This is different from atomicity violations where multiple threads enter the critical section, possibly leading to deadlocks and program hangs instead since no threads are then able to enter the critical section. To address this form of fault, we track which locks are owned by which threads. Upon exiting a critical section, if a lock is still owned by any thread no longer within the associated critical section, we can correctly identify the occurrence of an error, resulting in failure to release the lock. This occurs in stage **(3)** of Fig. 2.

A similar method is used for detecting errors within barriers as illustrated in Algorithm 2. Note that all updates to counters are performed atomically to avoid race conditions. By inserting functions directly before and after a barrier, we can identify when a thread enters and exits the barrier. If any thread attempts to exit the barrier before all involved threads have reached the barrier, it has violated the expected behavior of the barrier and is identified as an error. This requires knowledge of the number of threads that are involved in the barrier, which can be given as a parameter, collected from the initialization of the barrier object, or assumed by default to be the number of threads in use by the process. If the barrier in question can be encountered multiple times, additional checks are performed before Line 2 to ensure re-entering threads do not interfere with exiting threads.

Algorithm 2 Barrier Error Detection and Correction

```
Input: The instruction pointer $p$ for the barrier and the
    number of threads involved in the barrier $n$
1: Initialize entrance_counter to $n$, exit counter to 0
   When a thread attempts to enter the barrier:
2: Decrement entrance_counter
   When a thread attempts to exit the barrier:
3: if entrance_counter is not 0 then
4:     Roll thread back to the barrier and wait
5: else if exit_counter is not $n - 1$ then
6:     Increment exit_counter
7: else
8:     Reset counters to initial values
9: end if
```

B. Correction

Having detected the presence of errors, a thread can attempt local recovery via a rollback. This local recovery is similar to an aborted transaction in transactional memory systems. For barriers, rolling back is simple as errors are detected before threads can modify shared memory. When a thread is found to be exiting the barrier before all other threads have arrived, it is rolled back and forced to wait. When all threads arrive, the offending thread can then exit correctly together with all others. An example of both correct and erroneous executions is

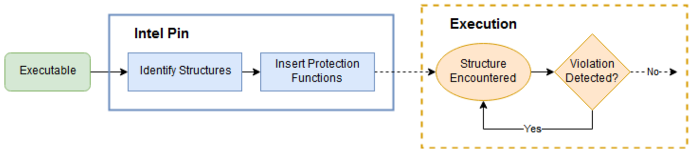

Fig. 4. Design work flow with Intel Pin.

shown in Fig. 3, where **(1)** and **(2)** mark the entry and the exit stages respectively. This reinforces the conceptual behavior of the barrier to ensure proper synchronization.

The process is somewhat more complex for locks as threads may modify shared variables. Additionally, we cannot be certain which thread within the critical section entered erroneously. Thus, both are rolled back and the logged writes are cleared, marked as **(2)** in Fig. 2. Only one thread has reached the point of modifying shared variables, so the rollback is relatively simple. This means that the log does not have to store backup values for a memory location for each thread. Since only one thread has been modifying shared variables, it can simply restore previous values while other threads reattempt the lock. Due to our method of conflict detection, we do not encounter situations where two threads can both modify a shared variable before the conflict is detected, which simplifies the logging and rollback processes. We recommend using re-entrant locks in conjunction with our system, as the correct owner will then be able to re-enter the critical section after a rollback. Upon successfully exiting the critical section, thread information is cleared from the log, allowing other threads to enter. Ultimately this mimics the conceptual function of the lock to ensure the correctness of the critical section. In summary, each of these correcting methods enforces the associated control structure behavior, thereby preventing the propagation of SDC.

C. Examples

For clarity, we provide two following examples to cover both thread and barrier encounters. Both examples will utilize two threads, thread **A** and **B**, to showcase the protection and correction mechanisms.

Locks: Suppose thread **A** encounters a lock at instruction p. The thread attempts to lock the lock, and our system attempts to claim ownership of the lock for thread **A** in line **1** of Algorithm 1. If thread **A** successfully claims ownership (line **2**), we assume thread **A** has appropriately locked the lock and can continue with the critical section (line **3**) while logging the usage of shared variables. Assume thread **B** encounters the lock while thread **A** is in the critical section. It is possible that thread **B** passes the lock entrance due to a fault in its or thread **A**'s locking process. Either way, the fault is detected in line **2** when thread **B** fails to take ownership. Both threads are then forced to reattempt acquisition of the lock in lines **5-7**, rolling back any changes made by thread **A**.

Barriers: Suppose thread **A** encounters a barrier expecting two threads at instruction p before thread **B**. When thread **A** enters the barrier to wait, marked as **(1)** in Fig. 3, *entrance_counter* is decremented from $n = 2$ to $n = 1$ as shown in line **2** of Algorithm 2. Thread **A** should not pass this barrier until thread **B** arrives to ensure proper synchronization. If an error occurs to cause thread **A** to pass the barrier early, the check in line **3** succeeds and line **4** is executed, forcing thread **A** back to waiting. When thread **B** arrives at the barrier, we again execute line **2**. Both threads attempt to exit the barrier, this time failing the check on line **3**. Both threads can thus exit the barrier, one executing line **5-6** to increment the *exit_counter* and the other simply exiting and resetting the initial values.

D. Implementation

For testing, we implement this method via Intel Pin as a Pin tool, which makes it flexible to work with any binary compiled for Intel processors. Intel Pin allows for both static and dynamic analyses and modifications of a program. As such, not only can we implement the transaction begin and end functions statically before execution, we are also able to track program locations and the control structure status dynamically during execution. Fig. 4 shows the overall workflow of the tool. Upon loading the binary, Pin applies the tools to the binary in two steps, called the instrumentation and the inspection passes. Instrumentation traverses the program statically to identify any instructions of interest, namely those related to the locks and barriers we aim to protect. It then inserts inspection functions into the binary that will be executed during run-time. At run-time, these functions intervene in the program execution, carrying out the logging methods as necessary to track the program status. Specifically, we locate every lock and barrier used by the program and add protection functions to each of them. These protection functions initialize the logging system with the program counter and thread information. This allows the logging system to detect the violations of the associated synchronization mechanisms.

Although our implementation is purely software, it could be augmented with hardware support. Our synchronization protection mechanisms need the additions of (1) an on-chip lock ownership directory, whose entries, say S[p], record the ID of the thread entering Lock p; see Algorithm 1, (2) an on-chip SRAM partitioned statically into zones, with Zone p for holding the log associated with Thread S[p], and (3) control logic for generating appropriate control signals and maintaining ownership directory entries. Both (1) and (2) are in the form of on-chip SRAM to improve performance. Additional instructions, similar to previous atomic instructions, could also be included to manage the lock ownership and logging operations involved with these added on-chip SRAM zones.

V. Evaluation

A. Vulnerability and Resilience

In order to test the effectiveness of our system, we execute the benchmark programs both with and without our protection mechanisms. All experiments have been run on a workstation

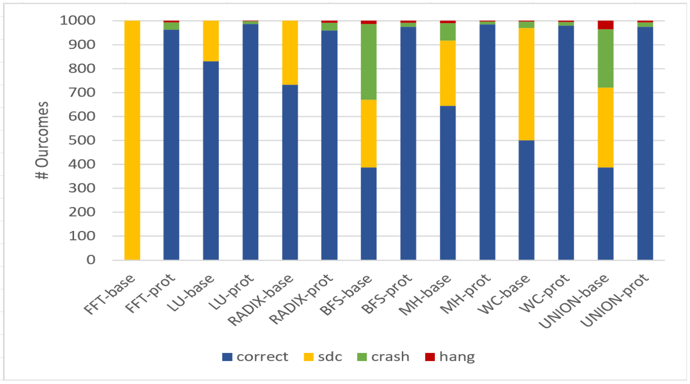

Fig. 5. Execution outcome breakdowns versus benchmarks under 64 threads.

with two Intel Xeon Platinum 8260 processors which support up to 48 threads each when enabling Hyperthreading, resulting in 96 total available threads. We test only up to 64 threads as some benchmarks require the thread count to be a power of two. As we have displayed in Fig. 1, higher thread count results in greater vulnerability, so unless otherwise noted, our experiments use the full 64 threads possible on our test machine. Once again, we use PINFI [8] to simulate transient faults by injecting one fault into each lock encountered during execution. For the following experiments, we restrict fault injection to both the synchronization code regions and our added protection code regions where applicable. We must inject into our added protection mechanisms to properly evaluate the vulnerability of the final system. This prevents full error coverage as the added code itself is vulnerable, although to a lesser extent.

The BigDataBench benchmarks and their categories [1] are listed in Table I. Specifically, we test the Fast Fourier Transform (FFT), LU matrix decomposition (LU), radix sorting (RADIX), graph operations (BFS), sampling operations (MH, Metropolis-Hastings implementation of the Markov chain Monte Carlo method), WordCount (WC), and set union (UNION). We have chosen not to test the Logical Operations category of the BigDataBench suite as its many samples are intrinsically sequential. Therefore, we have covered 7 out of 8 categories of parallel BigDataBench kernels. Both the baseline and protected versions of each benchmark are run through Intel Pin to provide a proper comparison between the two test cases with the maximum number of threads. The protection mechanisms are simply disabled in the baseline case. All benchmarks are executed with 64 threads. Our results are displayed in Fig. 5 and Table I.

As shown previously in Fig. 1, different benchmarks have different vulnerability profiles. FFT, LU and radix sort show varying degrees of SDC vulnerability, while the remaining four display considerable numbers of SDC and crashes. However, it is evident that the protection system removes almost all occurrences of crashes, hangs and SDC during the execution of these programs by correcting the soft faults within the control structures. It achieves a mean 93.6% error coverage across all

TABLE I
EXECUTION OUTCOMES FOR EACH BENCHMARK WITH (PROT) AND WITHOUT (BASE) OUR PROTECTION MECHANISM. A TOTAL OF 1000 TRIALS HAS BEEN EXECUTED FOR EACH BENCHMARK UNDER 64 THREADS. ER %: THE ERROR REDUCTION PERCENTAGE WHEN COMPARING THE NUMBER OF ERROR OCCURRENCES BETWEEN THE BASE AND PROT MODELS

Category	Benchmark	Outcome				ER %
		Correct	SDC	Hang	Crash	
Transform Operations	FFT-base	0	1000	0	0	
	FFT-prot	962	0	31	7	96.2
Linear Algebra	LU-base	831	169	0	0	
	LU-prot	986	0	12	2	91.7
Sorting	radix-base	733	267	0	0	
	radix-prot	959	0	32	9	84.6
Graph Operations	bfs-base	387	283	316	14	
	bfs-prot	974	0	17	9	95.8
Sampling	mh-base	645	272	73	10	
	mh-prot	984	0	13	3	95.5
Statistics Operations	wc-base	501	469	27	3	
	wc-prot	980	0	13	5	96.4
Set Operations	union-base	387	334	244	35	
	union-prot	975	0	18	7	95.9

kernels, with RADIX having the lowest coverage of 84.6% and WC having the highest coverage at 96.4%. We believe RADIX and LU show lower error coverage as they are already more resilient to errors and therefore there are fewer to correct. Interestingly, the protected benchmarks only contain crash and hang errors without any SDC occurrences. These crash and hang errors are preferable over SDC as they are more easily detectable and correctable during execution. Note that we do not claim that this method will address all possible errors that can occur in the program in general. Rather, the method focuses only on errors within the synchronization mechanisms, with errors beyond these synchronization structures deemed outside the scope of this work. By reducing these errors we prevent error propagation into other forms that may be more difficult or costly to detect with other methods.

B. Overhead

Execution Time Overhead To properly compare the cost of our system, we also investigate execution time overhead incurred by the implemented protection mechanisms on each benchmark under 4 to 64 threads. Both the baseline and the protected version are again executed through Intel Pin to provide an accurate comparison of the overhead caused by the system itself. We calculate the execution time overhead using $Overhead = \frac{T_{prot} - T_{base}}{T_{base}}$ where T_{prot} is the average execution time of 1000 trials of the protected benchmarks, and T_{base} is the average execution time of 1000 trials of the unprotected benchmarks. These results are shown in Fig. 6, where most benchmarks are found to have an overhead of less than 10% at all thread counts, with a mean overhead of at most 6.55% under 64 threads. These overhead levels are acceptable considering the improvement in error occurrences and the complete removal of SDC errors. It is clear that both BFS and LU have somewhat higher overhead than the others at

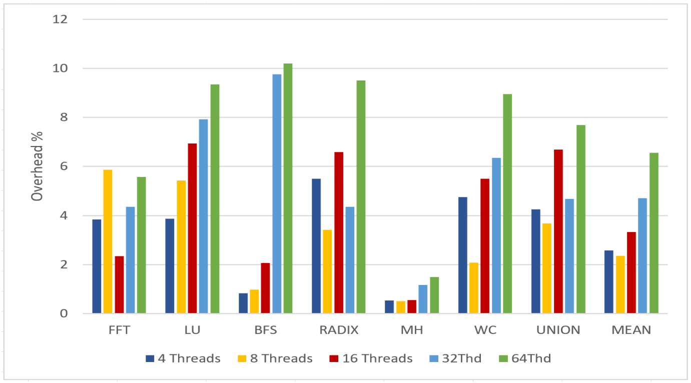

Fig. 6. Time overhead percentages for various benchmarks under 4 to 64 threads.

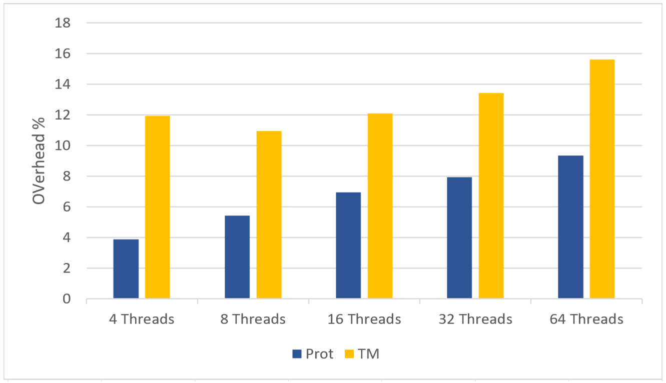

Fig. 8. Comparative time overhead percentages of our protection mechanism (prot) and transactional memory (TM) for LU under 4 to 64 threads.

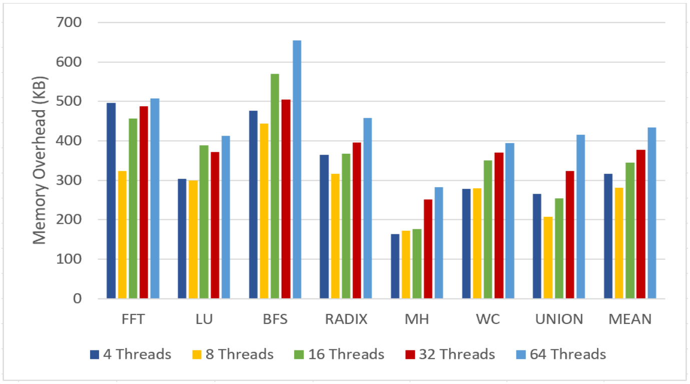

Fig. 7. Memory overhead for all benchmarks under 4 to 64 threads.

many thread counts. BFS is a larger application with greater memory requirements, leading to relatively higher overhead when experiencing context switching. Both BFS and LU also contain more complex and frequent concurrency control use, resulting in a higher accumulated overhead. As a result, it may not be wise to apply the protection mechanisms to every program; instead they should be applied on an algorithm-to-algorithm basis.

Memory Overhead For further evaluation of the system, we measure the memory overhead of our system on all benchmarks under 4 to 64 threads, as shown in Fig. 7. To calculate this, we record the memory high-water mark measured from within the program during execution. Note that the memory overhead is very small relative to the total amount of memory used by the programs. Most benchmarks use a maximum of 0.6-2.5GB memory during execution. As such, the overhead of 300-700KB is relatively negligible. As expected, we see larger overheads at higher thread counts.

C. Comparison with Transactional Memory

We additionally provide a comparison against complete software transactional memory systems, since our protection mechanism relies on similar checkpointing and rollback operations. Specifically, we test against the C++ atomic library and its included transactional memory interface. We do not compare with hardware transactional memory systems, which are incomparable to our software-based protection mechanism. Specifically, since we enforce the high-level behavior of locks, our system handles entire critical sections in addition to logging individual memory locations. As such, we can frequently detect conflicts when threads first enter a critical section rather than at every individual memory access. Given that our logging mechanisms are less intrusive than full transactional memory, they should therefore demonstrate lower overhead. To test this, we modified kernels for evaluation, with the results for LU and FFT shown in Fig. 8 and Fig. 9 respectively. Our baseline protects the shared variables using standard *pthread* locks and barriers for comparison against both our protection mechanism and the transactional memory implementation. We test the programs under 4 to 64 threads to gain insights into how the systems handle varying numbers of parallel agents. According to Figs. 8 and 9, the largest gap in execution time overhead overhead percentages occurs for FFT under 16 threads, with a difference of 10%. In total, we observe a geometric mean reduction of 47.3% and 57.5% in overhead for LU and FFT respectively. As we can see, the log-based protection mechanism consistently outperforms its transactional memory system counterpart at each thread count. While not displayed here, comparative overhead results for the remaining tested benchmarks exhibit similar performance gaps. This supports our previous claim that the protection mechanism is more lightweight than full transactional memory systems, resulting directly from simplifying many of the conflicts it must handle.

We also compare the memory usage for both our system and the transactional memory implementation, shown in Figs. 10 and 11. At all thread counts, our system consistently uses considerably less memory than the transactional memory implementation. Since our system can resolve conflicts sooner due to detecting the higher-level concurrency failures, it does not have to log as many values at one time for potential rollbacks, reducing the total memory used. Again, note that these values are still small relative to the total memory consumption of these benchmarks.

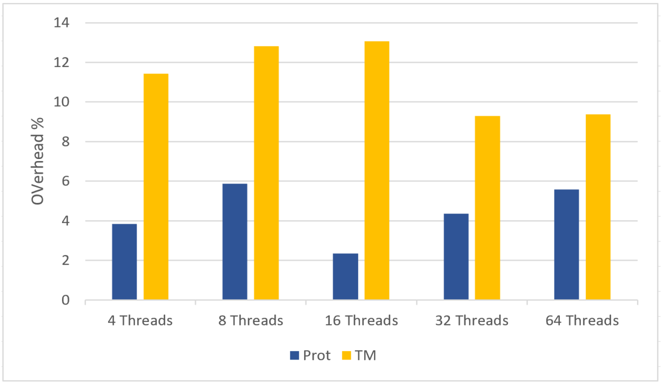

Fig. 9. Comparative time overhead percentages our of protection mechanism (prot) and transactional memory (TM) for LU under 4 to 64 threads.

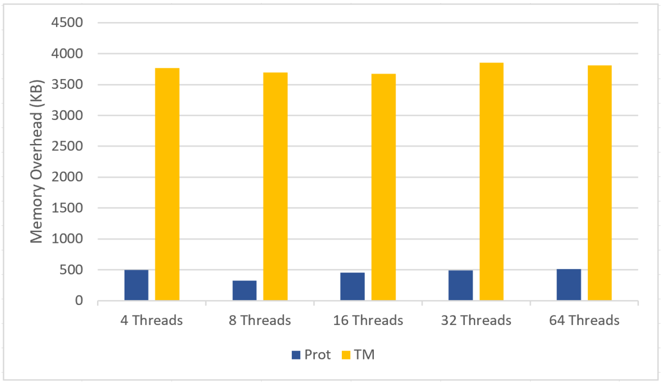

Fig. 10. Comparative memory overhead our of protection mechanism (prot) and transactional memory (TM) for LU under 4 to 64 threads.

VI. Conclusion

In this work we have presented a method for ensuring the correct and reliable operation of synchronization structures within parallel programs, specifically focusing on locks and barriers. By utilizing a logging system that tracks program locations, we can identify violations of these structures and recover from them locally, rather than requiring system wide checkpointing and recovery methods. Through our experiments, we demonstrate that this method can achieve a reduction in error of up to 93.6% for the representative BigDataBench kernels while maintaining acceptably low overhead, averaging 6.55% above the baseline. When compared with transactional memory, we find up to a 57.5% reduction in execution time overhead.

References

[1] BigDataBench Benchmark Suite. Available at https://www.benchcouncil.org/BigDataBench/index.html.
[2] Chen, S., and Peng, L. "Efficient GPU hardware transactional memory through early conflict resolution." Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2016.
[3] Elliott, J., Kharbas, K., Fiala, D., Mueller, F., Ferreira, K., and Engelmann, C. "Combining partial redundancy and checkpointing for HPC." Proceedings of the 2012 IEEE 32nd International Conference on Distributed Computing Systems. IEEE, 2012.

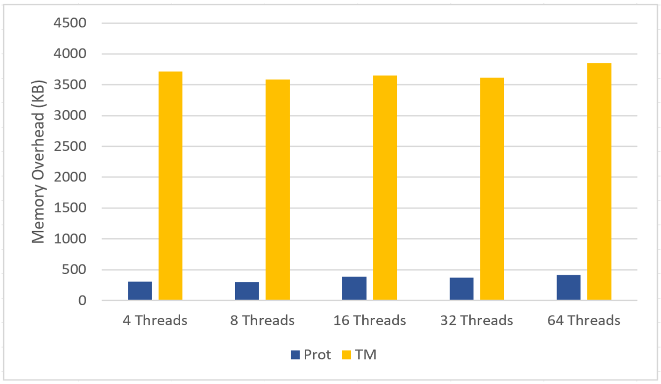

Fig. 11. Comparative memory overhead our of protection mechanism (prot) and transactional memory (TM) for FFT under 4 to 64 threads.

[4] Felber, P., Fetzer, C., Marlier, P., and Riegel, T. "Time-based software transactional memory." IEEE Transactions on Parallel and Distributed Systems 21.12 (2010): 1793-1807.
[5] Fung, W. W., Singh, I., Brownsword, A., and Aamodt, T. M. "Hardware transactional memory for GPU architectures." Proceedings of the 2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2011.
[6] Gabrys, Ryan, Eitan Yaakobi, and Lara Dolecek. "Graded bit-error-correcting codes with applications to flash memory." IEEE Transactions on Information Theory 59.4 (2012): 2315-2327.
[7] Intel Corporation, "Pin 3.2 User Guide." Available at https://software.intel.com/sites/landingpage/pintool/docs/81205/Pin/html/.
[8] Intel Pin Fault Injector (PINFI). Available at https://github.com/DependableSystemsLab/PINFI.
[9] Jangjaimon, I. and Tzeng, N.-F. "Effective Cost Reduction for Elastic Clouds under Spot Instance Pricing through Adaptive Checkpointing," IEEE Transactions on Computers, vol. 64, no. 2, pp. 396-409, February 2015.
[10] Joshi, A., Nagarajan, V., Cintra, M., and Viglas, S. "DHTM: Durable hardware transactional memory." Proceedings of the 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2018.
[11] LeCompte, T., Legrand, W., Chen, S., and Peng, L. "Soft Error Resilience of Big Data Kernels through Algorithmic Approaches," The Journal of Supercomputing, Vol. 73, pp. 4739–4772, Nov. 2017.
[12] Li, H., Chen, Z.m and Gupta, R. "Parastack: Efficient hang detection for MPI programs at large scale." Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. 2017.
[13] Sigdel, P. and Tzeng, N.-F. "Coalescing and Deduplicating Incremental Checkpoint Files for Restore-Express Multi-Level Checkpointing," IEEE Transactions on Parallel and Distributed Systems, vol. 29, no. 12, pp. 2713-2727, December 2018.
[14] Thomas, T. E., Bhattad, A. J., Mitra, S., and Bagchi, S. "Sirius: Neural network based probabilistic assertions for detecting silent data corruption in parallel programs." Proceedings of the 2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS). IEEE, 2016.
[15] Volos, H., Tack, A. J., Swift, M. M., and Lu, S. "Applying transactional memory to concurrency bugs." ACM SIGPLAN Notices 47.4 (2012): 211-222.
[16] Xiaoguang, R., Xinhai, X., Qian, W., Juan, C., Miao, W., and Xuejun, Y. "GS-DMR: Low-overhead soft error detection scheme for stencil-based computation." Parallel Computing 41 (2015): 50-65.