UNSERIALIZABLE INTERLEAVING DETECTION USING HARDWARE COUNTERS

Fernando Emmanuel Frati[1, 2], Katzalin Olcoz Herrero[3], Luis Piñuel Moreno[3], Marcelo Naiouf[1], Armando De Giusti[1,2]

[1] Institute of Research in Computer Science LIDI (III-LIDI),
School of Computer Science, UNLP, Argentina
{fefrati, mnaiouf, degiusti}@lidi.info.unlp.edu.ar
[2] National Commission of Scientific and Technical Research (CONICET)
School of Computer Science, UNLP, Argentina
[3] Group of Architecture and Technology of Computing Systems (ArTeCS),
Complutense University of Madrid, España
{katzalin, lpinuel}@dacya.ucm.es

ABSTRACT

Concurrent programs are needed to take advantage of multicore computers. Debugging such concurrent programs is very difficult due to their nondeterminism. So, error detection tools need to be used in production environments. One very popular detection technique is interleaving analysis, which detects atomicity violations in shared memory programs. Unfortunately, the algorithms that implement this technique can be very costly at runtime, restricting its use to the software testing stage. This paper shows how to use the hardware counters present in today's processors to detect the occurrence of unserializable interleavings. This optimization will reduce the overhead introduced by concurrency bug detection tools.

KEY WORDS

Parallel architectures, concurrent program, concurrency error, error detection, debugging, hardware counters

1. INTRODUCTION

In any concurrent program, the programmer has to specify how processes are synchronized. Depending on the communication model used, there are different methods to establish synchronization. For example, in a shared memory model, it is common to use semaphores or monitors, whereas in a distributed memory model, message passing is commonly used. Concurrency errors occur when the programmer makes a mistake when using any of these methods, resulting in race conditions, deadlocks, or atomicity violations.

One feature that makes concurrency errors particularly difficult to detect, is that they appear only under certain implementation condition, depending mainly on the non-determinism present in the execution order of processes. If these errors do not occur during the test, the program will be part of the production systems for which it was thought, making them vulnerable.

These errors, their causes, and ways to avoid them have been studied extensively by the scientific community. In 1978, Lamport established the concept of partial order between segments of processes [1] (it is called Happens Before relation), and it has been used to build race-condition detection tools. Lockset (proposed in 1997 [2]) is a different method to detect race conditions which verifies that all shared variables are protected at the time they are accessed. Deadlocks are another example of common mistake present in concurrent programs. In 1972, Holt [3] proposed a model that was able to detect deadlock conditions for shared resources (in fact, this technique is used by operating systems to manage their resources). Finally, with the advent of multiprocessors to conventional computers, atomicity violations have begun to take a leading role because their occurrence is more generalized than before. In 2006, Shan Lu [4] postulated an analysis of the order in which multiple threads access memory to help detect atomicity violations. The technique classifies interleavings as serializable or unserializable, where the latter may be atomicity violations (this issue is addressed further on this paper). Atomicity violations have also been addressed by other authors [5], [6].

Any of these errors can be found frequently in widely used, real programs, such as Apache, MySQL, or Mozilla Firefox [7]. Due to this non-determinism in parallel execution, it would be very useful to have tools to monitor applications in production environments.

Unfortunately, the overhead introduced by the detection algorithms is a determining factor in the viability of their use. For this reason, the current proposals tend to include a version of the algorithm which uses hardware extensions to accelerate them and reduce the impact of the instrumentation. These extensions involve changes to the architecture of the machines where they run (such as adding bits to the cache line).

In this paper, a different approach is proposed: using data available through the processor's performance-counters to choose when to run detection algorithms. Previous experiments [8] indicate that much of the overhead caused

by the monitoring tool is wasted in monitoring safe code regions. A similar technique was used by Greathouse in 2011[9] to optimize a tool for detecting race conditions called Intel Inspector XE. However, this tool is only able to detect race conditions, and it uses algorithms based on happens-before and lockset. The techniques and conclusions of this work are aimed at determining the feasibility of using counters to detect unserializable interleavings, which will optimize techniques for detecting atomicity violations in shared memory environments. The hypothesis is that it will reduce the instrumentation overhead by restricting access to those which are actually not serializable.

The paper is organized as follows: Section 2 offers a background and main concepts needed to follow the rest of the work. Section 3 presents our approach and the features addressed in this paper. Section 4 explains the algorithms used for the experiments. Section 5 describes the test environment. Section 6 shows the results of the experiments. Finally, in Section 7, conclusions are drawn.

2. BACKGROUND

2.1 Interleavings Analysis

Concurrency errors can be seen through the pattern access (reads and writes) that several processes make on each memory direction. When a variable is accessed between two accesses to it by a different thread, it is called interleaving. Depending on interleaving configuration, this could be a concurrency error. Table 1 shows the different possible configurations: in each case, both accesses by thread 0 are aligned to the left, whereas the one interleaved access by thread 1 is indented to the right. Over eight possible cases, four can be serialized. Serialized means that interleaving does not alter the perception that the processes involved have regarding the memory region they access, and is therefore considered to be safe. Interleavings 0, 1, 4 and 7 are safe because their occurrence produces the same effect as if they had not happened. On the other hand, cases 2, 3, 5 and 6 are

Case	Interleaving	Case	Interleaving

0	read
	  read
	read
		4	read
			  read
			write

1	write
	  read
	read
		5	write
			  read
			write

2	read
	  write
	read
		6	read
			  write
			write

3	write
	  write
	read
		7	write
			  write
			write

Table 1. Each case shows a different interleaving configuration. Access are interleaved read/write operations to the same memory direction by two threads.

interleavings that may have been caused by a concurrency error.

Proposals of this type seek the occurrence of unserializable interleavings. When an unserializable interleaving is detected, the tool emits an alert or initiates a corrective procedure (immediately after the error occurs), even to determine what instructions caused the error.

The instrumentation required to perform this type of analysis introduces a high overhead in the monitored-application's runtime. As mentioned in the introduction, algorithms to implement this technique include a version that uses hardware extensions to accelerate, and thus diminish the impact of instrumentation. Table 2 shows the results of running the software and hardware versions of AVIO [4] on the suite of benchmarks SPLASH-2 applications. Unfortunately these extensions involve changes to the architecture of the machines they run on (e.g., add bits to the cache line), their application in real production environments thus becoming non-viable.

Benchmarks	AVIO (Hardware)	AVIO (Software)
fft	0,5 %	42X
fmm	0,4 %	19X
lu	0,4 %	23X
radix	0,4 %	15X
Average	0,4 %	25X

Tabla 2. Overhead comparison between hardware and software versions of AVIO over SPLASH-2 suite benchmarks. X represents application runtime without instrumentation.

2.2 Hardware counters

All current processors have a set of special registers called hardware counters [10]. These registers can be programmed to count the number of times an event occurs within the processor during the execution of an application. Such events can provide information on different aspects of program execution (e.g., number of instructions executed, number of failures in L1 cache, or number of floating point operations executed). This information is intended to help developers understand the performance of their algorithms and optimize their programs. These events are poorly documented, and often vary between different architectures. Processor manufacturers usually offer a manual to indicate events and a brief description of what it is they are supposed to measure.

3. HARDWARE COUNTERS INTEGRATION WITH CONCURRENCY BUG DETECTION TOOLS

The alternative proposed in this paper is to use the information provided by hardware counters about what is happening in the processor to filter safe regions of code. It

will improve the performance of bug detection tools based on interleaving analysis. The key is keeping the monitoring tool disabled and toggling it to enabled mode if unserializable interleavings start occurring. In order to reach this we need:

- To know if unserializable interleavings can be detected through hardware counters.
- To determine whether most interleavings are serializable or not. This would hint as to whether there is a chance of optimization with our proposal.
- To characterize applications through their unserializable interleavings distribution in the code.

Each one of these features will be explained in the next subsections.

3.1 Unserializable interleaving detection through hardware counters

The goal is to find events that can be used to detect code segments of the program where unserializable interleavings occur. Since there are no events to measure interleavings, the problem is simplified to two accesses to the same memory address from different process. We assume each process to be running in different cores on a processor with at lest one level cache shared among them and a MESI based coherency protocol.

Each access can be a read or a write operation. The complete interaction between the accesses of each process is called access pattern. Table 3 summarizes the four possible cases. In all cases, the event must occur with each read or write operation.

In coherence MESI-based protocols, there is a "Shared state" (S) which indicates that the cache line has been read by more than one processor. The candidate events are those that indicate a transition to or from this state.

Each access pattern helps detect different interleavings. Table 4 shows the relationship between these cases. As it can be seen, the access patterns that are useful for the purpose of this study are 1 and 2, because they allow tracing unserializable interleavings 2, 3, 5, and 6.

3.2 Optimization feasibility

Our hypothesis is that overhead can be reduced if monitoring is restricted to the regions of code that are unsafe. Thus, to determine if the technique proposed can improve the performance of the bug detection tool with a particular application, first we calculate the ratio between the number of unsafe accesses and total accesses. An application with an unsafe access indicator (UAI) near to zero has a high potential because it means there are few unserializable accesses, whereas an application with an UAI near to one has a low potential.

Case	Pattern	Case	Pattern
0	read read	2	write read
1	read write	3	write write

Table 3. Each case shows a different pattern access to the same memory address by two threads.

3.3 Concentrated vs. distributed interleavings

Even if an application has a high optimization potential, it is important to know how these unsafe accesses are distributed by the code. If they are concentrated, it is more likely that our approach will reach a high performance improvement. But if they are scattered over the code, our tool could start continuously toggling between the enabled and disabled modes. This scenario is undesirable because it could affect not only the performance, but also the detection capability of the tool.

4. IMPLEMENTATION

4.1 Monitoring tool

In order to capture the information needed, we designed a software layer placed between the target application and the hardware. Basically, this tool detects when a new thread is created and sets its affinity to a specific core (we make sure each thread is running in different cores to take advantage of the changes in coherence states) and starts a hardware counter configured for the selected event. The tool also keeps software counters for program instructions. It can classify them in reads, writes,

Pattern access / Interleaving		0	1	2	3	4	5	6	7
		read read read	write read read	read write read	write write read	read read write	write read write	read write write	write write write
0	read read	YES	YES	-	-	YES	-	-	-
1	read write	-	-	YES	-	YES	YES	YES	-
2	write read	-	YES	YES	YES	-	YES	-	-
3	write write	-	-	-	YES	-	-	YES	YES

Table 4. Each cell shows if the pattern access can be useful to detect an interleaving.

branches, and others. This information will be enough to know the UAI of the target application. The tool has two operation modes:

1. Showing statistics collected when the target application finishes.
2. Showing partial statistics collected every preset number of instructions.

The second operation mode will allow us to determine the distribution of unserializable interleavings along the code.

4.2 Micro-benchmarks

To verify that an event can be used to distinguish between different access patterns, we designed the micro-benchmarks shown in Figure 1.

In each case, the simulated interaction between two threads is repeated *N* times (*N* is a parameter of the program). Each thread has a local variable called *tmp* used to simulate the reading operations. The shared variable is represented by an array of *N* elements *Var*, which is initialized by the main program thread. In order to have visibility regarding code operation, functions *P()* and *V()* were implemented with shared variables, instead of using pthread semaphores. Figure 2 shows the pseudocode for these operations.

Variables *s1* and *s2* act as semaphores initialized to 1 and 0 respectively. To avoid false sharing, a structure that fills the entire cache line for all vars (*s1*, *s2* and array elements) was designed. This allows analyzing state changes associated with the read/write operations on these variables.

5. EXPERIMENTAL SETUP

The monitoring tool was developed with the Pin [11] dynamic binary instrumentation framework, version 2.11. The perf_event system calls to access counters were used. Experiments were performed on a machine with two Xeon X5670 processors, each of which with 6 cores with HT. The operating system was a Debian wheezy/sid x86_64 GNU/Linux with kernel 3.2.0, and the compiler was gcc, version 4.7.0. No optimization options were used

```
P(cache_line *s){
  while(*s <= 0);
  *s--;
}

V(cache_line *s){
  *s++;
}
```

Figure 2. P() and V() functions used to synchronize threads.

because they may modify the codes to be analyzed. The micro-architecture of these processors, called Westmere, corresponds to the Nehalem micro-architecture with 32nm technology. This processor has three levels of cache – L1 (32KB) and L2 (256KB) are private of each core and L3 (12MB) is shared among all cores within the same physical processor [12]. The coherence protocol is based on MESI, but adds a fifth state, "Forward" (F), to indicate that the line was sent from one socket to another. In this scheme, a cache line that is read by several cores will be in state S for all copies and in all cache levels (L1, L2 and L3). When one of the copies is modified, its cache line state is changed to Modified (M). That change of state is propagated to the other cores, whose copies are changed to the Invalid state (I) [13].

The event selected was MEM_UNCORE_RETIRED: LOCAL_HITM, whose description in the Software Developer's Manual for Intel architectures [14] indicates that it counts the number of reading instructions on cache lines that are in state M in the cache of a sibling core, which means that this event can detect access pattern two (it occurs after the read operation). As far as we know, there is no event to detect access pattern one (at least in the Westmere micro-architecture).

To see if the event selected was appropriate for our purposes, the micro-benchmarks explained in Section 4.2 were used. In all remaining experiments, the SPLASH-2 benchmark suite [15] was used. These benchmarks were run on 2 threads using the test inputs. Table 5 shows the configuration for each program of the suite.

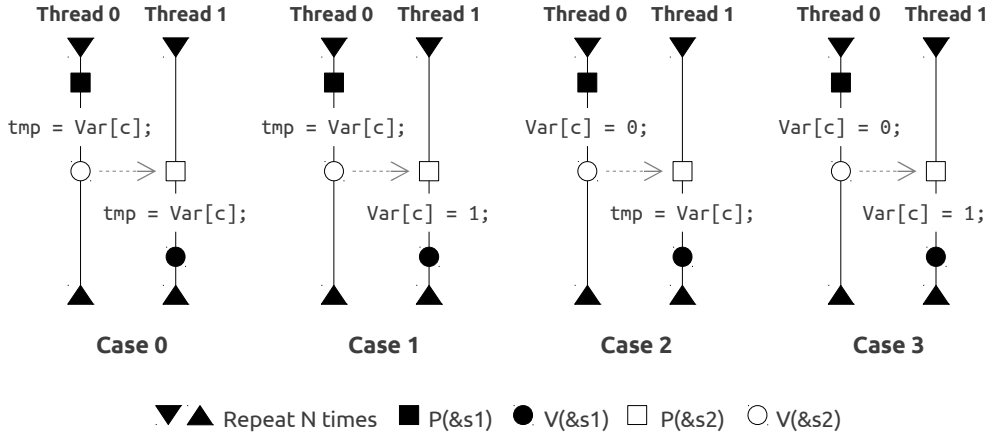

Figure 1. Micro-benchmarks for each access pattern.

Application (configuration)		RD+WR	Others	Total
BARNES (16K particles)		1138,19	718,85	1857,04
CHOLESKY (Tk15.0)		183,06	242,15	425,22
FFT (64K points)		9,26	24,53	33,79
FMM (16K particles)		437,37	2160,04	2597,41
LU (512x512 matrix, 16x16 block)	Continuous Block	152,64	221,75	374,39
	Non Continuous Block	2,51	3,4	5,91
OCEAN (258x258)	Continuous Partition	160,08	265,09	425,18
	Non Continuous Partition	148,13	273,97	422,1
RADIOSITY (room -ae 5000.0 -en 0.050 -bf 0.10)		575,93	1021,19	1597,13
RADIX (256K keys, 1024 radix)		27,82	29,13	56,94
RAYTRACE (car)		379,43	576,67	956,09
VOLREND (head)		2108,2	2107,58	4215,78
WATER (512 molecules)	NSQUARED	195,84	382,36	578,19
	SPATIAL	177,73	318,93	496,67

Table 5. Millions of instructions by application.

6. RESULTS

To reduce the possibility of error in the measurements, each test was run ten times and the results averaged.

6.1 Unserializable detection

This experiment was designed to demonstrate that the event selected can track access pattern 2, which will allow detecting unserializable interleavings 2, 3 and 5. Since the micro-benchmarks explained in section 4.2 use a shared var to synchronize threads, an average of one event triggered by each iteration is expected (remember that each event is triggered with a read operation after a write operation in cache of a sibling core). The only difference is found in case 2, where thread 1 must trigger an extra event after each iteration caused by a reading operation after a writing operation on variable *Var[c]*.

The results of the experiments can be seen in Table 6. Each micro-benchmark was executed with different sizes of *N* to show the behavior of the event scales with the size of the problem.

Pattern	Thr	10000		100000		1000000	
Case 0	0	11807,7	1,18	101987,2	1,02	1009867,8	1,01
	1	11138,1	1,11	98839,6	0,99	980244,2	0,98
Case 1	0	11635,8	1,16	94526,7	0,95	926031,5	0,93
	1	11058,3	1,11	98601	0,99	974063,8	0,97
Case 2	0	12420,8	1,24	97233,9	0,97	936528,9	0,94
	1	20114,6	2,01	192174,5	1,92	1904729,5	1,90
Case 3	0	11654,7	1,17	95997,3	0,96	920952,7	0,92
	1	10791,3	1,08	95941,7	0,96	894380,1	0,89

Table 6. Results of evaluating each access pattern between two threads for different problem sizes.

The table should be interpreted as follows – rows show the event count for thread 0 and thread 1 in each access pattern by each size problem, and calculate the ratio between this value and the size of the problem (cells highlighted in gray). Results are rounded to two decimal places. It should be noted that each access pattern exhibits the same behavior regardless of problem size. In all cases, with each iteration of the loop, the event is triggered once. This can be explained from Figures 1 and 2. For operations in which the read/write activity occurs in the proper order, threads are synchronized using two shared variables, *s1* and *s2*. For example, thread 1 keeps reading *s2* until thread 0 increases its value (see the gray arrow in Figure 1).

Because the write policy is a write-allocate one, the processor running thread 0 brings variable s2 to its own cache. Then updates the coherence state to M (modified). After that, thread 1 reads the line in M state from the core cache where it is running thread0, triggering the event. The same applies to thread 0. As we mentioned before, it can be seen that for case 2, thread 1 has counted an extra event with each iteration, showing that the event can be used to distinguish pattern 2 from the others. Thus, the selected event will identify three of the four unserializable interleavings. As previously mentioned, an event for pattern 1 was not found. However, it has to be considered that, if interleavings are concentrated in the code, there is a high probability that most of the times the tool will be enabled when interleaving case 6 happens. This hypothesis will be tested in future experiments.

6.2 SPLASH-2 unsafe access indicator

The monitoring tool was used with each application of the SPLASH-2 suite benchmarks. Only read and write instructions were considered because the technique is based in interleaving analysis and does not require any other type of instructions. Results are shown in Table 7. In most of the cases, the UAI is lower than 1%. This means that 99% of the read and write operations do not trigger the event and, in consequence, we know for sure that there are no unserializable interleavings in cases 2, 3 or 5 in those code regions. It should be noted that UAI is not an indicator of the number of shared instructions; it only serves as a measure of the number of unserializable interleavings. The program may have a higher percentage of read and write operations to shared vars, but these may be serializable or case 6 interleavings.

6.3 Unserializable interleavings distribution

As mentioned in Section 3.3, the UAI is not enough to know if our approach could improve tool performance with the application. In order to decide if the monitoring tool can be disabled, we need to know how interleavings are distributed along the code.

These experiments were made with the monitoring tool in operation mode 2, as described in Section 4.1. The tool was set to take approximately 100 samples for each

Application		Thread 0			Thread 1		
		Event Count	RD+WR (M)	UAI	Event Count	RD+WR (M)	UAI
BARNES		471613	578,21	0,08%	479037	575,49	0,08%
CHOLESKY		349135	116,02	0,30%	274857	83,59	0,33%
FFT		52355	5,26	0,99%	38218	4,01	0,95%
FMM		274997	221,66	0,12%	235089	220,14	0,11%
LU	Continuous Block	46271	80,94	0,06%	39468	72,42	0,05%
	Non Continuous Block	33724	1,36	2,49%	27294	1,16	2,35%
OCEAN	Continuous Partition	554190	81,20	0,68%	528390	81,32	0,65%
	Non Continuous Partition	514282	74,55	0,69%	461943	74,59	0,62%
RADIOSITY		886884	298,43	0,30%	903262	296,89	0,30%
RADIX		51664	14,46	0,36%	46883	14,47	0,32%
RAYTRACE		563691	243,62	0,23%	586121	141,93	0,41%
VOLREND		304107	1670,51	0,02%	737558	472,17	0,16%
WATER	NSQUARED	189560	99,02	0,19%	155233	97,01	0,16%
	SPATIAL	254036	88,95	0,29%	140784	88,83	0,16%

Table 7. Unsafe Access Indicator (UAI) estimation for each application of the SPLASH-2 suite. Counts are expressed in millions of instructions.

application. The tool has a set of software counters for each thread; every time an instruction is executed, the corresponding counter is increased. When a new thread is created, its counter is initialized with the partial count of thread 0. When the counter reaches the sample step preset for the application, the hardware counter is read and saved with the software counter number in a file. The hardware counter is set to 0 and the process continues until the application finishes.

Figures 2 and 3 show the distributions for three applications of the suite. These cases were chosen because they are representative of the rest. The horizontal axis corresponds to the number of instructions executed, and the vertical axis to the count of events. Threads 0 and 1 are the blue and red lines, respectively. The legend corresponding to the horizontal axis was intentionally left blank because large numbers make it difficult to read the figure. As shown in the figure, three different behaviors were found among the programs:

– In FFT, each thread has several peaks of the event higher than zero. It should also be noted that there is a significant amount of code that can be executed

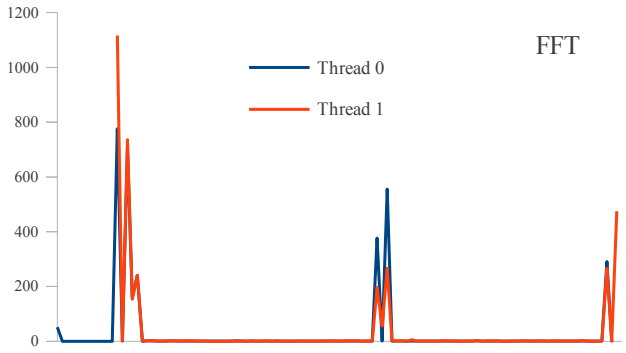

Figure 2. Ideal unserializable interleaving distribution .

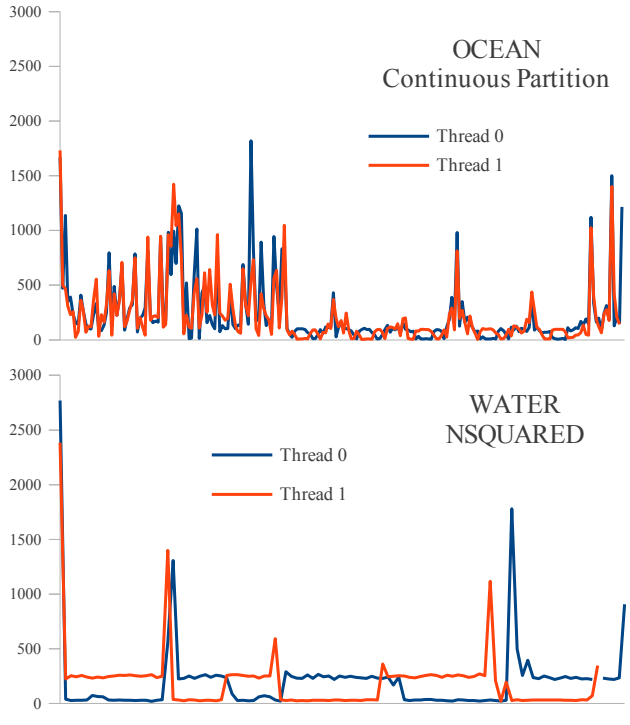

Figure 3. Special unserializable interleaving distribution

without instrumentation: it is where both threads have zero count of the event between the peaks. BARNES, CHOLESKY, and RADIX behave in the same way.

– In OCEAN, there seems to be significant issues between the threads. This is the worst-case scenario for our proposal, because an application with this behavioral pattern would cause our tool to constantly toggle between the enabled and disabled states.. Both OCEAN, both LU, VOLREND, and RAYTRACE versions showed this behavior.

– WATER presents an interesting situation – even though the program registers counts higher than zero all the time, there is a pattern in the way events occur. Most of the time, event count is lower than 300. This kind of behavior is probably caused intentionally by the programmer, such as the case of the synchronize mechanism through shared vars we used for the micro-benchmarks. FMM, RADIOSITY, and WATER SPATIAL have this behavior.

It should be noted that a program can exhibit one or all of these behaviors in different runs. For this reason, our goal was to show that a dynamic technique for detecting unsafe code regions can be implemented, rather than obtaining static information on program execution.

7. CONCLUSION

This paper presents a model for using hardware counters in detecting unserializable interleavings. Through the use of instrumentation techniques, we were able to establish a relationship between memory access patterns, the size of the problem, and the proposed event. From the experiments, it can be concluded that counters can be used to detect the execution of at least three of the four cases of unserializable interleavings.

Our experiments show that the technique proposed can help dynamically identify unsafe code regions. Depending on the way the program has been developed, it is possible to use this information to decide when to disable or enable the monitoring tool. In some cases, where there are constant unserializable interleavings along the code, our proposal could be useful if trained first to adjust to the threshold indicator of unsafe interleavings (in this paper, it was assumed to be zero).

We are currently working to use this information to develop a smart tool to detect atomicity violations, which would only be active on those program segments that run unserializable interleavings, which is expected to achieve an improvement in execution times of detection algorithms.

Recalling that AVIO introduces a 24X penalty in average, this optimization could significantly reduce the overhead of the software version without modifying the architecture of the target machine.

REFERENCES

[1] L. Lamport, Time, clocks, and the ordering of events in a distributed system, *Communications of the ACM*, vol. 21, pp. 558–565, jul-1978.

[2] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, Eraser: a dynamic data race detector for multithreaded programs, *ACM Trans. Comput. Syst.*, vol. 15, nº. 4, pp. 391–411, nov. 1997.

[3] R. C. Holt, Some Deadlock Properties of Computer Systems, *ACM Comput. Surv.*, vol. 4, nº. 3, pp. 179–196, sep. 1972.

[4] S. Lu, J. Tucek, F. Qin, and Y. Zhou, AVIO: detecting atomicity violations via access interleaving invariants, *SIGPLAN Not.*, vol. 41, nº. 11, pp. 37–48, 2006.

[5] B. Lucia, J. Devietti, K. Strauss, and L. Ceze, Atom-Aid: Detecting and Surviving Atomicity Violations, in *ISCA '08: Proceedings of the 35th Annual International Symposium on Computer Architecture*, Washington, DC, USA, 2008, pp. 277–288.

[6] B. Lucia, L. Ceze, and K. Strauss, ColorSafe: architectural support for debugging and dynamically avoiding multi-variable atomicity violations, *SIGARCH Comput. Archit. News*, vol. 38, nº. 3, pp. 222–233, 2010.

[7] S. Lu, S. Park, E. Seo, and Y. Zhou, Learning from mistakes: a comprehensive study on real world concurrency bug characteristics, *SIGARCH Comput. Archit. News*, vol. 36, nº. 1, pp. 329–339, 2008.

[8] F. E. Frati, K. Olcoz Herrero, L. P. Moreno, D. M. Montezanti, M. Naiouf, and A. De Giusti, Optimización de herramientas de monitoreo de errores de concurrencia a través de contadores de hardware, in *Proceedings del XVII Congreso Argentino de Ciencia de la Computación*, La Plata, 2011, vol. XVII, p. 10.

[9] J. L. Greathouse, Z. Ma, M. I. Frank, R. Peri, and T. Austin, Demand-driven software race detection using hardware performance counters, *SIGARCH Comput. Archit. News*, vol. 39, nº. 3, pp. 165–176, jun. 2011.

[10] B. Sprunt, The basics of performance-monitoring hardware, *IEEE Micro*, vol. 22, nº. 4, pp. 64– 71, ago. 2002.

[11] 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 *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, New York, NY, USA, 2005, pp. 190–200.

[12] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, Intel Corporation, Manual 248966-026, abr. 2012.

[13] D. Levinthal, *Performance Analysis Guide for Intel® CoreTM i7 Processor and Intel® XeonTM 5500 processors*, Intel Corporation, Report Version 1.0, 2009.

[14] *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Intel Corporation, Manual 253669-043US, may 2012.

[15] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, The SPLASH-2 programs: characterization and methodological considerations, *SIGARCH Comput. Archit. News*, vol. 23, pp. 24–36, may 1995.