Thesis Overview:

Software for Multi-Core Processor-Based Architectures.

Automatic Detection of Concurrency Errors.

Fernando Emmanuel FRATI

School of Computer Science, National University of La Plata, Argentina
PhD Thesis in High Performance Computing
Advisors: Armando E. DE GIUSTI, Marcelo NAIOUF, Katzalin OLCOZ HERRERO
{fefrati, degiusti, mnaiouf}@lidi.info.unlp.edu.ar, katzalin@ucm.es

March 2015

All commercially available processors (even the processors used in mobile devices) have the typical multicore architecture (Yeap, 2013) – the shared memory programming model dominated over the sequential programming model as the optimal way for obtaining maximum performance offered by these architectures. Execution order assumptions between instructions and atomicity when accessing legacy variables from the sequential programming model are no longer valid in the new model, whose implicit non-determinism when running concurrent programs forces programmers to use some synchronization mechanism to make sure these properties are present. Frequently, programmers make mistakes when synchronizing the processes, which results in new programming errors such as deadlocks, race conditions, order violations, simple atomicity violations, and multivariable atomicity violations. These errors cannot be detected by traditional debugging methods, so tools that can help detecting and correcting them are required.

The main objective of this paper is to propose a software implementation model for concurrency error detection tools that allows reducing process overhead without decreasing its detection capacity. The general model proposed uses software dynamic instrumentation in such a way that an analysis routine can be activated from a signal generated by a hardware event that indicates the possibility of an error occurring. The results obtained showed that, for the case study (an atomicity violation detection algorithm called AVIO), the version that uses the model proposed can detect the same bugs as the original version, but in only 25% of the time (in average) required by it.

This Thesis (Frati, 2015) is organized in eight chapters and five appendixes with supplementary material related to the topics discussed in the body.

In **Chapter 1,** an overview of the topic is presented, as well as the main objective of the thesis, its specific objectives, and the research methodology to be used. The methodological design is based on the research process, where specific objectives guide the process through the different levels of knowledge and achieving the last specific objective results in achieving the main objective.

In **Chapter 2,** the reference theoretical framework on program debugging is introduced. This framework provides a description of the characteristics that make concurrency errors unique in nature, and determines the reasons why traditional debugging methods are not suitable for these errors. It was found that, even though deadlocks and race conditions have gained popularity in the scientific community, only 29.5% of the reported errors correspond to deadlocks, while the remaining 70.5% are other types of errors. Among the errors that are not deadlocks, *atomicity violations represent more than 65% of the total*, *96% occur between two threads* and *66% involve a single variable (*Lu, Park, Seo and Zhou, 2008). After this stage, it was decided that the case study used to guide the rest of the work would be a simple atomicity violation.

In **Chapter 3,** current techniques and methods proposed by the scientific community are compared, applied to atomicity violation detection and correction. AVIO (Lu et al, 2006) is found to be the tool with the best performance and detection ability that can be fully run on software. Since its source code is not available and there are no commercial implementations of this tool, we developed our own implementation. This implementation was validated based on detection ability and performance versus the data presented by the authors in their original work. The results show that our implementation is equivalent to AVIO, since the results reported with this tool were successfully replicated. Even though AVIO is better than previous proposals, the overhead it introduces (25x in average) is too high to be used in production environments.

In **Chapter 4,** the problem to be studied is delimited, and the advantages and disadvantages of the methods used in the previous proposals are discussed. Error detection methods use program dynamic instrumentation at the level of the statement because software implementations require the addition of calls to an *analysis routine* for each statement that accesses memory. The main cause for the overhead is identified to be *instrumentation granularity*, since it penalizes execution time in more than one order of magnitude. As a consequence, the problem to be solved is limited to reducing the overhead produced by statement-level instrumentation. It was observed that the possibility of error occurs only during statements that access shared data, and it was determined that there is an opportunity to optimize the instrumentation process if it is restricted to shared memory accesses only. The best option to detect the exact time when these accesses take place is when there are changes in the cache memory that is shared among the cores that run the processes.

In **Chapter 5,** the use of hardware counters is introduced to detect non-serializable interleavings (interleavings that cause atomicity violations). This resource was previously used to optimize a race condition detection tool (Greathouse et al, 2011). *Hardware counters* are a collection of special records that are available in all current processors (Sprunt, 2002). These records can be programmed to count the number of times that an event occurs in the processor while an application runs. The events provide information about different aspects of the execution of a program (e.g., the number of statements executed, the number of L1 cache failures, or the number of floating point operations executed). The discussion in this chapter allowed establishing that, even though there is no single event that can indicate the occurrence of interleaving cases, these can be represented through memory access patterns. As a consequence, accesses to shared data when accessing information stored in the cache coherence protocol can be detected.

In **Chapter 6,** the main contributions of this thesis work are developed. The design of a general dynamic instrumentation model is proposed that allows enabling an analysis routine (an atomicity violation detection algorithm in our case study) from a signal that is generated by a hardware event that indicates the possibility of an error occurring. To achieve this, an event validation method had to be defined so as to ensure that any candidate events were appropriate for the purpose of our work. This validation involved analyzing event and error frequency and distribution during the execution of various applications. After the suitability of the selected event was established, experiments were run with the counters under an operation mode called *sampling* (Weaver, 2014), which allows configuring counters to generate signals for a process if an event occurs. Thus, programmers indicate the number of events that should occur before the signal is triggered, adjusting this value based on application requirements. This operation mode was used to decide when to enable the analysis routine of the detection tools to reduce code instrumentation. On the other hand, since counters cannot be configured to send a signal in the case an event *does not occur,* the use of a *timer* is proposed to verify at regular time intervals (*sampling interval*) whether it is safe to disable the analysis routine (e.g., because there were no atomicity violations detected throughout the last interval).

In **Chapter 7,** a possible implementation is shown for the proposed model using the AVIO atomicity violation detection tool, resulting in a new version of this tool called AVIO-SA (Share-Aware AVIO). The efficacy of the model proposed is assessed based on overhead and detection ability. When monitored applications are started, AVIO-SA's analysis routine is disabled. The moment an event is detected, the routine is enabled and remains active for a while, as in the original version of AVIO. Eventually, AVIO stops detecting interleaving events and the analysis routine is disabled. It was found that a sampling interval of 5ms allows AVIO-SA detecting approximately the same number of interleaving events as AVIO, but with a significantly lower execution time. To complete performance tests, experiments were carried out with HELGRIND (a race condition detection tool), and the overhead of each tool for each application was estimated. In average, HELGRIND had an overhead of 223x; AVIO had an overhead of 32x; and AVIO-SA, one of 9x. As regards error detection ability with AVIO-SA, experiments were carried out with known bug kernels as well as with applications with real bugs (Apache) and reported bugs results were compared between AVIO and AVIO-SA (from SPLASH-2). AVIO-SA successfully passed all tests.

Chapter 8 presents the conclusions of the thesis. The results obtained showed that the new version can detect the same bugs as AVIO but using (in average) only a fourth of the time required by the original version to do so; therefore, the general objective of the work is considered to have been achieved. In conclusion, since AVIO-SA makes fewer changes in the monitored application execution history, this is a better option to be used in production environments. Finally, the following new lines of work following up on the work presented in this thesis are listed:

- Feasibility for applying the model to algorithms that detect other types of errors.
- Reliable comparison methods for error detection proposals.
- Tools for forcing potentially defective histories.

- Alternative methods for reducing instrumentation overhead.
- Dynamic instrumentation and parallel application profiles.

Fernando Emmanuel Frati
fefrati@lidi.info.unlp.edu.ar

References

Frati, F. E. (2015). Software para arquitecturas basadas en procesadores de múltiples núcleos. Detección automática de errores de concurrencia. (Tesis). Facultad de Informática. Retrieved from http://hdl.handle.net/10915/44643

Greathouse, J. L., Ma, Z., Frank, M. I., Peri, R., y Austin, T. (2011). Demand-driven software race detection using hardware performance counters. SIGARCH Comput. Archit. News, 39 (3), 165–176. Downloaded from http://doi.acm.org/10.1145/2024723.2000084

Lu, S., Tucek, J., Qin, F., y Zhou, Y. (2006). AVIO: detecting atomicity violations via access interleaving invariants. SIGPLAN Not., 41 (11), 37–48. doi: http://doi.acm.org/ 10.1145/1168918.1168864

Lu, S., Park, S., Seo, E., y Zhou, Y. (2008). Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. SIGARCH Comput. Archit. News, 36 (1), 329–339.

Sprunt, B. (2002). The basics of performance-monitoring hardware. IEEE Micro, 22 (4), 64–71.

Yeap, G. (2013, diciembre). Smart mobile SoCs driving the semiconductor industry: Technology trend, challenges and opportunities. En Electron devices meeting (IEDM), 2013 IEEE international (pp. 1.3.1–1.3.8). doi: 10.1109/IEDM.2013.6724540

Weaver, V. (2014, abril). Manpage of PERF_event_open. Downloaded on July 22, 2014 from http://web.eece.maine.edu/~vweaver/projects/perf_events/perf_event_open.html