System Responsiveness Verification of large Multi-Processor System Configurations using Micro-Benchmarks and a Multi-Level Analysis

Dr. Ralf Winkelmann[1], Edward Chencinski[2], Hanno Eichelberger[3], Michael Fee[4], Carsten Otte[5], Christoph Raisch[6]

[1] IBM Systems, Böblingen, Germany, +49 7034 643 0686
[2] IBM Systems, Poughkeepsie, USA, +1 845 435 8227
[3] IBM Systems, Böblingen, Germany, +49 7031 16 1654
[4] IBM Systems, Poughkeepsie, USA, +1 845 435 6834
[5] IBM Systems, Böblingen, Germany, +49 7034 2744267
[6] IBM Systems, Böblingen, Germany, +49 7031 16 458

***Abstract*- Computer system development necessitates design changes to incorporate new technology. These changes can have unintended negative side effects, such as statistically significant system responsiveness issues. We architected, implemented and executed a flexible environment that enables us to identify and address these issues in pre- and post-silicon simulation environments (software simulator, hardware accelerator, real hardware). A deterministic Linux kernel based execution environment has been created to implement and run micro-benchmarks targeting multi-processor coordination test scenarios. Sophisticated multi-hierarchy analysis and predication tooling were invented. We successfully applied this method to multiple generations of IBM Mainframe systems creating a benchmarking history, thereby providing insights for the creation of significant system level hardware improvements.**

I. INTRODUCTION

Computer system development necessitates design changes to incorporate new technology. These changes can have unintended negative side effects. Industry trends of higher on-chip integration increases communication time between components in our systems. The trend of diminishing transistor performance scaling together with increasing computing demands necessitate increasing numbers of processors to deliver the necessary overall system capacity growth.

Design efforts to improve performance must mitigate tradeoffs made in packaging designs that can impact latencies. Even when using the same packaging, faster chip technology will increase latencies as there are more, smaller processor cycles in the same path. Mitigation techniques include the incorporation of caches, asymmetrical nodal system structures, more aggressive prefetching and out-of-order processing in the CPUs (Central Processing Units), also referred to as processor cores or simply just 'cores'. The success of systems developed & released under this general trend supports this approach. But this approach carries numerous hazards with it, particularly within system processes that require unavoidable CPU contention.

Multiprocessor coordination requires sequences that provide systemic contention and can potentially span the entire system. Furthermore, these operations interrupt processors that have speculatively run-ahead thereby causing damaging retry sequences. The repetitive nature of computing, combined with these speculative, run-ahead designs and longer distribution times can lead to system wide instability. Use of these coordination sequences should therefore be minimized. When they must be used, techniques to reduce their impact must be implemented and verified pre-silicon.

In this paper, we describe a methodology that enables us to implement, run, analyze and debug multiprocessor coordination sequences in pre-silicon and post-silicon environments. Pre-silicon experiments are carried out on cycle accurate hardware description language (HDL) based models that include the full richness of our scale-up Mainframe computers, including multiple cores, chips, nodes, and drawers [1]. Furthermore, the software, i.e. the operating system and the test cases, are the same across pre-silicon and post-silicon experiments. Therefore, pre-silicon results predict post-silicon results with high accuracy and allow us to find and correct hardware deficiencies prior to building hardware.

Our goals are:

- Analyze and debug system responsiveness behavior in a pre-silicon environment with high accuracy.
- Find and correct design issues in the hardware model in a HDL during the implementation phase that result in sub-optimal system responsiveness behavior.
- Optimize multi-processor coordination code sequences, including new specialized hardware features.
- Provide best practices for system level software developers, i.e. firmware and operating systems.

II. BACKGROUND

Accelerator-based Simulation

Accelerated simulation is executed on a hardware apparatus composed of a profusion of specialized ASIC processors. This hardware apparatus can simulate numerous basic logic components in parallel. The hardware design is therefore synthesized for executing on those accelerators [4]. Accelerated simulations are much faster than software-based simulation. The main bottleneck of accelerator-based simulation is the communication between accelerator and host.

System Simulation

System simulation ensures that all units of the system work together correctly. Each unit is verified separately in 'unit' and 'element' models and then finally integrated into a system model. During system simulation or verification all units are verified working together [4]. Based on such system simulation, key performance metrics can be derived, which come very close to those of the final system.

III. METHODOLOGY

Overview / System Design Level

We architected, implemented and executed a flexible environment that enables us to address the system responsiveness problems mentioned above, both pre-silicon and post-silicon. In the following section, we discuss requirements that significantly influenced the architecture of our environment.

The overall complexity of scale up computer systems continues to increase on all levels. The end of frequency scaling together with the desire to improve compute performance results in more complex system designs. While there is still opportunity to improve performance within components of the system, more and more features are implemented across multiple components which are intended to operate cooperatively to further increase system performance. This trend requires more mature pre-silicon system level testing. System level testing to ensure architecture correctness has been done for decades now [4] and general performance measurements and prediction are also already in place.

The multi-processor coordination features that we address with our methodology strongly depend on system components playing well together in concert. It is not possible to accurately verify the performance of these features separately within each component anymore. The critical behavior occurs via interaction between the components on the microarchitectural level. Therefore, models on higher abstraction levels do not work either. With this in mind, a "full" size cycle accurate hardware model has been chosen for the pre-silicon environment.

A "full" size hardware model is already challenging by itself. However, the systems running in production mode at customer sites have a complex software structure running on top of the hardware. Figure 1 illustrates the full stack.

As of today, it is beyond our capabilities to run the full stack in a cycle accurate simulation environment. We have other environments that address other aspects of integration testing on the full stack [3].

Our work requires the ability to analyze and debug on the microarchitectural hardware level. Therefore, we decided to build an environment that runs a Linux operating system bare metal on the underlying hardware. Another alternative would be to run our test cases without operating system support on the hardware. However, this would result in a significant increase in the test case complexity, because some basic support typically provided by operating systems is required. In section 'Execution Environment' we describe the design decisions that resulted in a robust and flexible framework.

Running Linux bare metal on our hardware is supported for both pre-silicon and post-silicon execution with only minor changes. This is of importance when pre-silicon results are expected to predict post-silicon results, albeit with less clarity due to limits of the pre-silicon model size. On pre-silicon experiments we are constrained by model size and simulation speed. When real hardware becomes available, we rerun the suite of pre-silicon experiments and validate that both match. Then, we extend the testing to evaluate the larger state space by varying parameters, such as, the number and physical span of CPUs, variations in the algorithms and other parameters. While the pre-silicon

experiments enabled us to find and correct deficiencies in the hardware implementation and study algorithm early on, the post-silicon experiments allow us to do a full characterization.

Test case turn-around time must be limited to practical levels in the pre-silicon environment. In pre-silicon environments, we are constraint in two dimensions: Model size and simulation speed. Both parameters influence the turn-around time. Therefore, partial system configurations are chosen that still represent the complexity of the system to a large extend. The test case design must take the simulation speed into consideration. While classical performance evaluations on real hardware tolerates experiments that run for hours and days, our capabilities on accelerators are in the range of seconds.

Finally, we need to be able to debug the results of our experiments. Therefore, we built a hierarchical approach that enables us to analyze the system efficiently on various level of abstraction. In the pre-silicon environment, we can debug on any level. Cycle accurate data on the implementation (HDL) level is available. Debugging on higher abstraction levels is shared across pre-silicon and post-silicon experiments, however, a richer set of tools and data is available for pre-silicon experiments. In this paper, we give an overview of our debugging methodology.

Figure 2 is illustrating a high-level overview of the pre-silicon accelerator based environment.

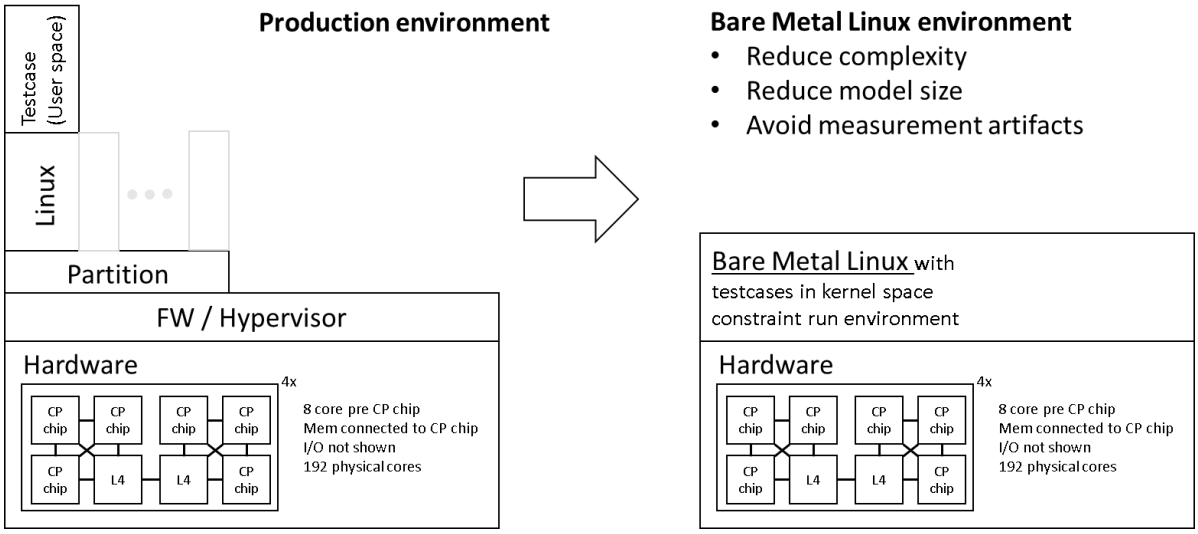

Figure 1. The Figure on the left side illustrates an example of a production environment on a mainframe computer. Real customer environments are usually more complex, but here we use a less complex example to show the difference to our Bare Metal Linux environment. For our low-level analysis and debug of system responsiveness micro-benchmarks we need to run on a cycle accurate model to study the interaction between code and hardware in detail. Reducing the complexity in the environment and in the model size is a key enabler for success. Therefore, the Firmware code and the hypervisor are not part of our environment (see Figure on the right side). Instead, Linux runs directly on the underlying hardware.

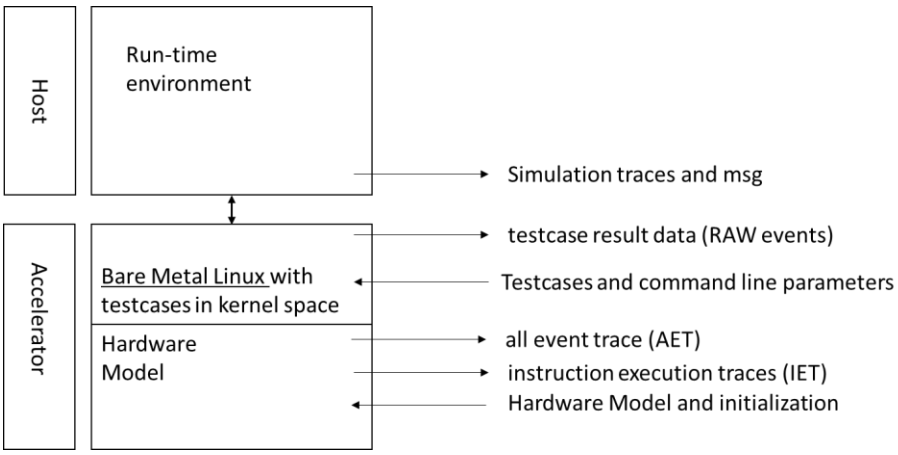

Figure 2. The pre-silicon simulation environment is shown. On the accelerator, the hardware run with cycle accuracy. The Bare Metal Linux along with the test cases are loaded into the system memory in the accelerator and are execute from there. On the host run-time support for loading the model into the accelerator, initializing it, monitor facility, and controlling the execution is performed. Selected inputs and outputs that are relevant for our purpose are shown on the right-hand side of the figure.

Execution Environment

In the past, many low-level functional and performance tests have been implemented as simple, directed instruction stream testcases written in assembler code. Some of them even evolved into tiny operating systems with their own support infrastructure for console I/O, etc. The advantage of that execution environment is close control over the hardware, and thereby results are accurate and 100% reproducible. These test cases are, however, limited to the functional aspects that they implement.

In contrast, some test frameworks make use of operating systems such as Linux. Such tests benefit from the operating system's support infrastructure. Each test has a full featured memory manager, console I/O, a storage subsystem, and IP networking at its disposal. The downside, however, is that the test cases in these environments lose some control over the machine state to the operating system they run on. The results may vary due to side effects outside of the control of the test case, such as timer events or I/O interrupts.

We have created a test infrastructure which allows us to get all the benefits from both worlds without having any of those downsides. Our test infrastructure is implemented as a kernel module that is loaded into Linux. It provides a user interface that utilizes *debugfs*[1] which takes care of parsing the input and starting the corresponding test case, which in turn is implemented in another kernel module per test case.

Each test case has two phases. During the first phase, the Linux infrastructure can be used to allocate memory and gather other resources needed for test execution. Our test infrastructure provides a library of function calls that assist during this phase, such as the possibility to allocate buffers for test data which can be accessed via *debugfs* files. It allows us to register callback functions along with parameters which are scheduled to be executed during the second phase of the test case execution. Once the test case is done with this stage, it calls our test infrastructure to enter the second phase.

The infrastructure now utilizes *stop_machine*[2], which is also used to get the machine into a specific state during other critical operations in the kernel such as changing the internal linkage tables during load or unload of kernel modules. *stop_machine* triggers a multi-stage process that monopolizes all available processors, disables kernel preemption and closes the interrupt masks of all processors. Once this has been performed on all processors, regular Linux features such as memory allocation are no longer available and the machine is in a highly predictable, and closely controlled state. The test infrastructure now runs all scheduled exerciser functions on each CPU, and waits for them to complete.

[1] Debugfs on Wikipedia: debugfs is a special file system available in the Linux kernel since version 2.6.10-rc3. It was written by Greg Kroah-Hartman. debugfs is a simple-to-use RAM-based file system specially designed for debugging purposes. It exists as a simple way for kernel developers to make information available to user space.
[2] Stop_machine: https://github.com/torvalds/linux/blob/master/kernel/stop_machine.c

During these test cases, we chose to have a three-step approach:

- first all data buffers that store measurement data are preloaded into each CPU's close caches by writing to them sequentially.
- each CPU hits a synchronization point that ensures that all processors have completed their preparations. This synchronization point is implemented via a memory location and atomic updates.
- Now all CPUs do the test case payload synchronously, and record their results into per-CPU buffers that were prepared previously.

After completion of a test case, we leave *stop_machine*. This in turn enables interrupts, enables kernel preemption and lets all CPUs run their respective work. We can now use user-space tools to work with the data generated via our *debugfs* interface.

Our test cases are computed both on real hardware, and on simulated circuits. On simulated hardware, the amount of CPU cycles available is restricted. Processing results inside the simulator is therefore not practical. To get hold of the resulting data, we've implemented a *debugfs* interface that exports the absolute addresses of each data buffer along with its length. A small tool is used inside the simulator to extract these memory addresses and export them to an environment outside the simulator, which does have access to the simulated memory. From there we extract the data at the full native speed of the simulating machine.

Micro-Benchmarks Deep Dive

Background:

The goal of the subject hardware locking verification effort is to prove that the hardware design enables consistently efficient and effective execution of code employing hardware locks to control cooperative CPU execution of workloads across many cores. The verification of combined Software/Hardware locking implementations in a 100-core real world physical and distributed system is a difficult, complex problem with a large design space. Typically, hardware engineers try to navigate verification of this design space through theoretical high level models which unfortunately can't reproduce all non-typical cases and event alignments occurring in real world combinations of complete middleware stacks. Furthermore, running these complete middleware stacks in full system HDL simulation requires a significant amount of CPU cycles, which results in extremely long runtimes. Because of the difficulty of meeting the locking hardware verification goals with such approaches, we developed a suite of critical microbenchmarks containing (among other things) typical locking scenarios from these workloads which can be runtime configured to represent different locking cases.

This section describes the synthetic microbenchmark which was implemented in this project to measure the system responsiveness of different hardware logic options. The focus of the presented work is the performance and statistics of locking or semaphore behavior.

Testcase description:

This test case executes several parallel threads. These threads are synchronized via semaphore locks. Every thread tries to be the first thread to store (a one) into the semaphore lock. After the thread stores into the lock it executes a critical section for a specific time. During this time, all other threads are prevented from storing into the lock through hardware interlocks of the semaphore design. Furthermore, the test case prevents them from executing their critical section during this time. After the active thread finishes its critical section, it runs into an idle section. During this idle time this thread does not try to store into the lock and run its critical section again. Using such idle time allows a better synchronization of threads (described below). This testcase structure allows efficient, rapid coverage of the critical locking scenarios and the micro-benchmarking enables rapid determination of the efficiency and effectiveness of the hardware that implements locking.

Listing 1 shows the implementation of the lock latency scenario in pseudocode. The tester can specify the count of iterations (see line 1). The thread starts to run a Load-and-Test (LTG) instruction. This instruction reads the content of the lock and checks if the lock is held by another thread. For reading the lock, the cache line which contains the lock is requested from the cache hierarchy. If the lock is not held by another thread, then the Compare-and-Swap (CSG) is executed (line 4). This instruction checks if the lock is still free and stores into it to reserve the lock. Therefore, an exclusive request of the cache line in the cache hierarchy is required. Afterwards, the thread executes the critical section for a specific time (line 6). When the critical section is finished, the thread stores a zero into the lock. Unlocking is implemented in this way. The thread waits for a specified time until it starts with a new iteration. The tester can specify the execution of some line access noise during this idle time. This noise can be specified to be either *store* or *load* noise. These values are stored or loaded into the same cache line which contains the lock. With

every request to the cache line which contains the lock, a request to the cache hierarchy considering this line is executed. This noise results in a less efficient access of the cache line by others threads. Such noise may be unintended consequences of real program code.

```
Require
  CRITSEC                    Duration of critical section
  ITERATIONCOUNT             Count of iterations
  IDLETIME                   Idle duration after critical section
  NOISE_LOAD, NOISE_STORE    Specifies the noise
  lock                       Variable which represents the lock
  n_start, tmp               Temporary variables

1     for i=0 to ITERATIONCOUNT do
2           rc=LTG;                                    // RDVAL
3           if rc==1 goto 2;
4           rc=CSG;                                    // LOCKED
5           if rc==1 goto 2;                           // CSFAIL
6           wait(CRITSEC);
7           lock=0;                                    // UNLOCKED
8           n_start=time();
9           if NOISE_LOAD:
10                while (time()-n_start<IDLETIME) tmp=lock;
11          else if NOISE_STORE:
12                while (time()-n_start<IDLETIME) lock=tmp;
13          else:
14                wait(IDLETIME);
```

Listing 1. Pseudo code for lock latency test case.

Figure 3 shows the state diagram which is executed by this test case. The start of an iteration is declared as TRYLOCK. It is followed by a read of the lock (RDVAL). If the read to the lock is successful, the thread compares the lock again and tries to store a 1 into the lock (LOCKED). If the compare fails, it jumps back (CSFAIL). Afterwards, it executes the critical section for a specific duration (CRITDONE). The critical section is finally finished by storing a 0 into the lock (UNLOCKED). Finally, an idle time is executed, which can execute additional noise. This flow is executed until the amount of iterations is finished.

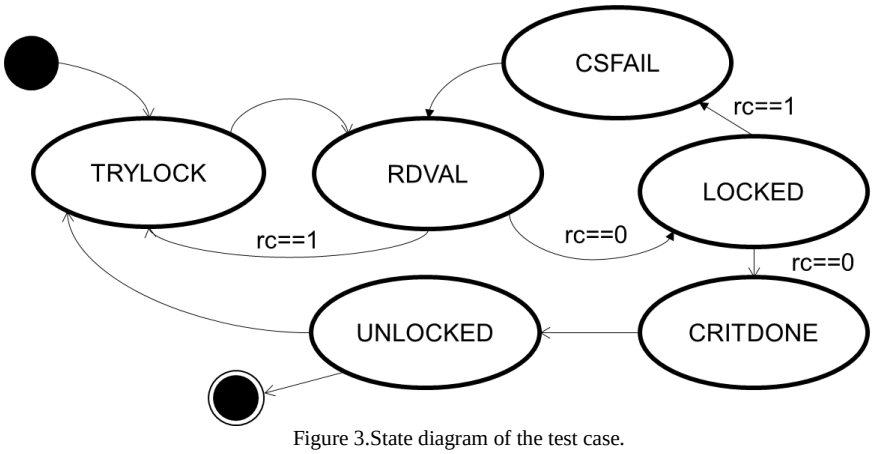

Figure 3.State diagram of the test case.

The test case can be executed with different parameters. Table **1** describes the most important parameters. The parameter *CPUCount* defines how many threads are instantiated by the test case. The developer can decide how many iterations every thread should run, by defining the *Iterations* parameter. A higher iteration count results in a bigger data set and shows a better representation of the system behavior. However, in simulated environments a high iteration count can cause long simulation runs. The *CritSec* parameter defines the duration of the critical section. The *IdleTime* after the critical section can be parametrized as well. Long idle times cause a better synchronization of the threads

(defined as ‘burst mode’). A short idle time causes a kind of fight between the threads to get to lock during the whole run, which we define as ‘storm mode’.

Table 1. Most important parameters for locking test case.

Parameter	**Description**
CPUCount	This number defines the count of threads which are instantiated
Iterations	This number defines the count of iterations
CritSec	This time defines how long the thread holds the lock
IdleTime	This time defines how long the idle time is executed
Noise	This parameter defines which type of noise is executed

The locking micro-benchmark seems to be very simple. However, it results in complex behavior on the micro-architectural hardware level. Our system has special hardware features to optimize lock efficiency. This micro-benchmarks enabled us to perform high quality analysis of those features and therefore further optimize our system.

Test Case Execution and Automation

In the previous section the locking micro-benchmark was described and a list of selected parameters was introduced. When varying any of those parameters a multi-dimensional space to be analyzed is opened. Given the number of parameters we can easily create more than a thousand different configurations to be run.

We invented tooling that allows us to constrain the parameter variation and therefore only create a relevant subset of combinations to be run.

Tooling is also required to automate the selection of CPUs within the system. Our systems are characterized by a non-uniform distance between processors. Several cores are packaged on a processor chip, multiple processor chips form a node, multiple node are gathered in a drawer and the entire system consists of multiple drawers. The topology does not only result in differences in latency between pair of CPUs, but also brings variations in routing. Those aspects have a measurable impact on locking behavior and are important to be considered when choosing CPUs. Instead of choosing CPUs by ID, we introduced abstractions, such as, near, distributed and dedicated.

In practice, when running on real hardware, we apply few constrains and obtain from a couple of hundreds to more than a thousand different combinations. This multi-dimensional analysis turned out to be very useful to develop a more precise understanding of the system behavior.

In an accelerator based system simulation we are constraint by model size and simulation speed. Therefore, we only run a subset of experiments compared to a real hardware environment.

Tooling is independent of whether we run pre-silicon or post-silicon.

Analysis Toolchain

Functional Characteristics

After the initial test runs in this project the team learned that simple statistical analysis of average and deviations does not help to gain deeper insight into the measurement data. Hardware Engineers need to evaluate how alterations of system design influence overall locking behavior, especially if new secondary effects are unintentionally caused by the changes. To them the most useful graphical tools were event delta distributions and hardware location transition heatmaps, both of which can be used to identify unexpected behavior without a deep statistical knowledge.

As mentioned previously the overall test case runtime environment eliminates all side effects as much as possible, which also means individual CPU core events get recorded in PU local buffers without synchronizing to other CPUs. The current System Z implementation has a fully synchronized cycle accurate clock locally available on all processors, which is added to each recorded event (defined as time-of-day, TOD). Therefore, as a first step of the analysis toolchain, a single ordered system-wide view of all events is generated. Further steps in the analysis then identify individual time deltas between matching events on different cores and generate time delta histograms. For example, matching events could be the following: Core A frees a lock in memory location and core B successfully acquires the lock in memory. Transition heat maps then show a 2-dimensional distribution of how often core A was the previous owner of a lock when core B successfully acquires it.

Non-functional Toolset Requirements

The scope of this project is planned for multiple years. Tooling developed for this project will be reused and modified vs rewritten on each future generation. Therefore, the team made a conscious decision to pick a development

environment which will still be available and maintained in the future. Initially coding started in a combination of C++, python and gnuplot and is currently migrated to spark with gnuplot as the diagram tool. Spark itself is used in a 'pandas data frame-like' approach, as that subset can be easily understood by technical engineers with background in matlab, numpy or similar tools. Another important aspect to consider is the readability of code as undetected errors introduced in analysis will lead to wrong evaluations of hardware design options which will lead to wrong conclusions for the overall system implementation.

In addition, everything used in generating the data, analyzing the data and controlling execution is tracked in a single previsioned source code repository. Therefore, results are exactly reproducible over time. We can determine which environment generated which results over many years.

Typical Usage Scenarios

In a typical usage scenario, a developer would generate a set of measurement points, each consisting of the event series of all associated processors. For real systems, the required storage can get into the 100Gigabytes to capture atypical outliers with some probability. After generating this data, it would be transferred to an analysis server (or laptop) to batch process the raw data into system-wide time series and produce all desired diagrams for the critical measurement points. Next, it is verified that the diagrams confirm the expectation of system behavior; When behavior fails to match expectations, then the process will drill down into individual sequences of events and generate system traces (see next section).

Debugging

This section describes how the verification engineer analyses and debugs degraded responsiveness based on the performance diagrams. It describes the method of hierarchical debugging which yields insights much faster than observing the events in the simulated hardware for long time periods.

The raw data output of the analysis toolchain includes a sequential **trace of state machine events**. Table 2 shows an example trace. The first column in this list defines the core number. The second column shows the stored TOD. The third column contains information about the iteration and additional event data. The fourth column shows the state machine event type. The following table shows an example of a state machine event trace. In this example the two different cores access the cache line with a RDVAL (row 2 and 3). However, only core 003 gets the LOCKED (row 4). Thus, the CSG instruction of core 001 results in a CSFAIL (row 5). In our example, the verification engineer wants to analyze the root cause of the CSFAIL and why core 001 makes an RDVAL before core 003 gets the LOCKED state.

Table 2. Example trace of statemachine events.

Core ID	Time of Day (TOD)	Iteration Data	Event type
000	0x0000000011ddfd72	0x0800000000000001	UNLOCKED
003	0x0000000011ddfef8	0x0200000000000000	RDVAL
001	0x0000000011ddff64	0x0200000000000000	RDVAL
003	0x0000000011de0052	0x0400030000000001	LOCKED
001	0x0000000011de010a	0x0300030000000001	CSFAIL

If inconsistencies of events are detected in the list of state machine events, this area can be analyzed in more detail generating a **trace of executed core instructions**. Therefore, the TOD of state machine events is translated into simulation cycles. Using these cycle times the simulator can be instructed to generate a trace of all core instructions. The input for this is a starting time and an end time for the trace. For each core a trace of instructions is generated. Every entry in this trace includes the starting and ending cycle of the instruction, the instruction name and the absolute target address. The following Table 3 shows a trace for the LTG and the CSG of core 003. This core has the cache line with the lock in a read-only cache and executes LTG (row 1). This way, the other thread 001 can access the line as read-only to execute LTG as well. For the CSG instruction the core is required to access the cache line exclusively. Therefore, it sends a request for exclusive access to the cache hierarchy (row 2). Afterwards, core 003 can execute the CSG and store into the cache line (row 4). The STCKF instruction implements the TOD request for the event tracing (row 3).

Table 3. Example trace of core instructions.

Thread	Start -> End Sim Cycle	Operation	Address
T0	000289107195->000289109219	LTG	abs=0000000001273A00
T0	000289107219->000289109319	EX Response	<0000000001273A00>
T0	000289107597->000289109769	STCKF	
T0	000289107599->000289110361	CSG	abs=0000000001273A00

Using this debug data, the problem can be isolated. However, the logic designer needs a trace of all signal events in the system to analyze and detect the exact root cause. Such a trace is called a **trace of signal events**. Using this trace, every signal event in the simulated hardware can be observed. It can be used for manual debugging by the logic designer. Figure 4 shows a screenshot of signal events in an appropriate viewer. Note that it can be seen that this traces shows the progressing of the time of day, for example.

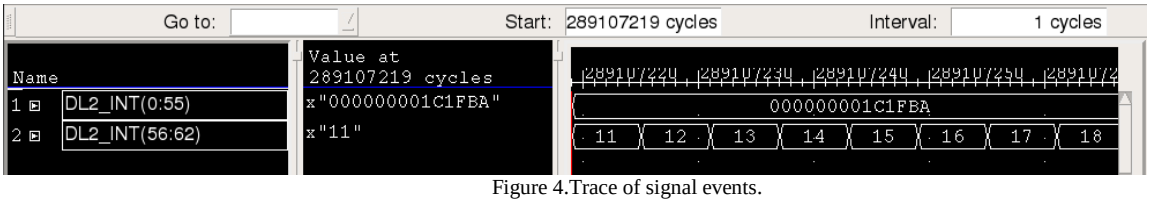

Figure 4.Trace of signal events.

Using this method of debugging, the root cause of the wrong behavior detected in state machine events can be efficiently isolated. The generation of sequential state machine events is a result of postprocessing and does not required additional monitoring. The generation of instruction sequences is faster than tracing long periods of the system operation including every signal. However, it provides enough information to determine the failing cycle range. A detailed signal trace can be generated for the determined range. This detailed trace can be used by the designer to debug the root-cause.

IV. RESULTS

Our methodology has been applied to the two last generations of mainframe systems that are on the market. In addition, it is used for the development of upcoming systems. In this section, we provide three different results that were selected based on z13 hardware experiments.

A recommendation to always use the Load-and-Test loop before the Compare-and-Swap

In principal, there is no functional requirement to insert a Load-and-Test (LTG) instruction prior to the Compare-and-Swap (CSG) instruction when using locks to coordinate multiprocessor activities. However, the combination of LTG and CSG takes advantage of micro-architecture features in our mainframe hardware to yield a superior result for the program.

In this section, we illustrate the benefits by providing simulation results from z13 real system runs.

The following results are based on a 128 CPU configuration. CPUs are spread out across the system on 4 drawers, the critical section time is 4 micro seconds, with no sharing of the cache line that holds the lock variable. We've used burst mode timing in this example.

The graphs below show the distribution of time it takes from one CPU obtaining the lock until the next CPU obtains the lock (locked-to-locked). The X-Axis shows the time in micro-seconds for the locked-to-locked transition. Y-Axis shows the frequency of this time's occurrence across the set of lock negotiation iterations. There are two pairs of graphs. The upper one is in logarithmic scale on the Y-Axis and the lower one is showing the same data, but in a linear scale.

The first pair of graphs (see Figure 5) illustrates the locked-to-locked distribution for just the CSG, that is, with no LTG before the CSG. The second pair (see Figure 6) shows the distribution with LTG before the CSG.

The optimal locked-to-locked time is 4 micro seconds, since the critical section time is 4 micro seconds. In Figure 6 the peak is a little bit above 4 micro seconds. This is close to optimal. In Figure 5 the peak is shown at 10 micro

seconds. All locked-to-locked values that are larger than 10 micro seconds are collapsed into the 10 micro seconds bar. Thus, there are values that fall far above 10 micro seconds.

This clearly demonstrates that the LTG before the CSG is a must have, especially when intending to run on a larger number of CPUs that are physically dispersed across the drawers.

A Comparison of near vs distributed CPU placement

Our systems are characterized by a non-uniform distance between processors. Several cores are packaged on a processor chip, multiple processor chips form a node, multiple node are gathered in a drawer and the entire system consists of multiple drawers. The topology not only results in differences in latency between pairs of CPUs, but also brings variations in routing. In Figure 7 and Figure 8 the impact on locking efficiency due to CPU placement is illustrated. In both experiments 32 CPUs on z13 ran with a critical section time of 4 micro seconds. The idle time has been set to a large value, so that there is no interference between iterations. A total of 2000 iterations was run for each experiment.

While the graphs in Figure 7 are based on near CPU placement, graphs in Figure 8 are based on distributed placement. 'Near placement' refers to CPUs that are close to each other within the same package. Therefore, the distance between CPUs is minimal, as is the latency of signal communication. 'Distributed placement' refers to the case when the distance between CPUs is maximized. For System Z that means cross drawer.

As expected there is a clear benefit in locking efficiency for near placement.

The Impact of False Sharing

When talking about false or unintended sharing, we are referring to loads and stores from inside the non-critical section to the cache line in which the lock variable resides (see section 'Micro-Benchmarks Deep Dive'). We have three different flavors: none, load or store. The two diagrams in Figure 9 and Figure 10 show locked-to-locked data for 32 CPUs on z13 applying 'near placement'. The critical section time is 4 micro seconds.

Without false sharing (see Figure 9) the system behaves very well. With false sharing (see Figure 10) the efficiency drops significantly.

This data makes it clear that false sharing has a huge impact on overall locking performance! Therefore, it is highly recommended that false sharing is avoided whenever possible.

V. SUMMARY AND FUTURE WORK

In this paper, we described a methodology and environment that enables us to efficiently implement, run, analyze and debug multiprocessor coordination sequences in pre-silicon and post-silicon simulation environments. This is done via software simulator, hardware accelerator and real hardware.

The environment has matured beyond its original research or prototype state. It is now integrated into our product development process. The results from testing in this environment influence system design decisions as well as provide pre-silicon detection and correction of hardware implementation flaws. Furthermore, these results allow the identification and documentation of best coding practices for firmware, middleware and application software developers.

Future Work – In addition to the analysis of locking behavior, there are multiple other scenarios that can be categorized into the system responsiveness class. The interaction of I/O, processors and different levels of caches is one example. Therefore, we are extending our suite of micro-benchmarks to cover numerous other scenarios within this environment.

REFERENCES

[1] C. R. Walters, P. Mak, D. P. D. Berger, M. A. Blake, T. C. Bronson, K. D. Klapproth, A. J. O'Neill, R. J. Sonnelitter, V. K. Papazova , "The IBM z13 processor cache subsystem", IBM Journal of Research and Development (Volume 59, Issue 4/5), pp. 3:1-3:14, 2015

[2] D. F. Ackerman, M. H. Decker, J. J. Gosselin, K. M. Lasko, M. P. Mullen, R. E. Rosa, E. V. Valera, and B. Wile, "Simulation of IBM Enterprise System/9000 Models 820 and 900", IBM Journal of Research and Development (Volume 36, Issue 4), pp. 751-763, 1992.

[3] J. von Buttlar, H. Böhm, R. Ernst, A. Horsch, A. Kohler, H. Schein, M. Stetter, and K. Theurich, "z/CECSIM: An efficient and comprehensive microcode simulator for the IBM eServer z900", IBM Journal of Research and Development (Volume 46, Issue 4/5), p. 607-615, 2002.

[4] B. Wile, W. Rösner, J. Goss, "Comprehensive Functional Verification: The Complete Industry Cycle (Systems on Silicon)", ASIN:B00193EG5C, 2005.

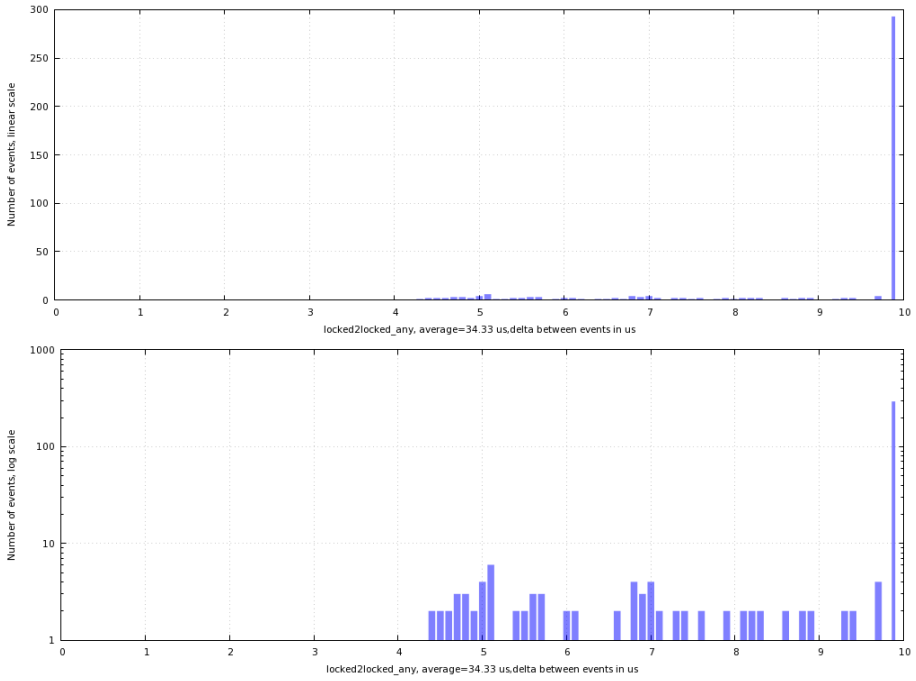

Figure 5. Distribution for Locked-to-Locked on z13. Without the Load-and-Test before the Compare-and-Swap most of the time it takes more than 10 micro seconds to move the lock from one CPU to the next. The best case would be 4 micro seconds (critical section time). In Figure 6 the Load-and-Test precedes the Compare-and-Swap and dramatically improves the performance.

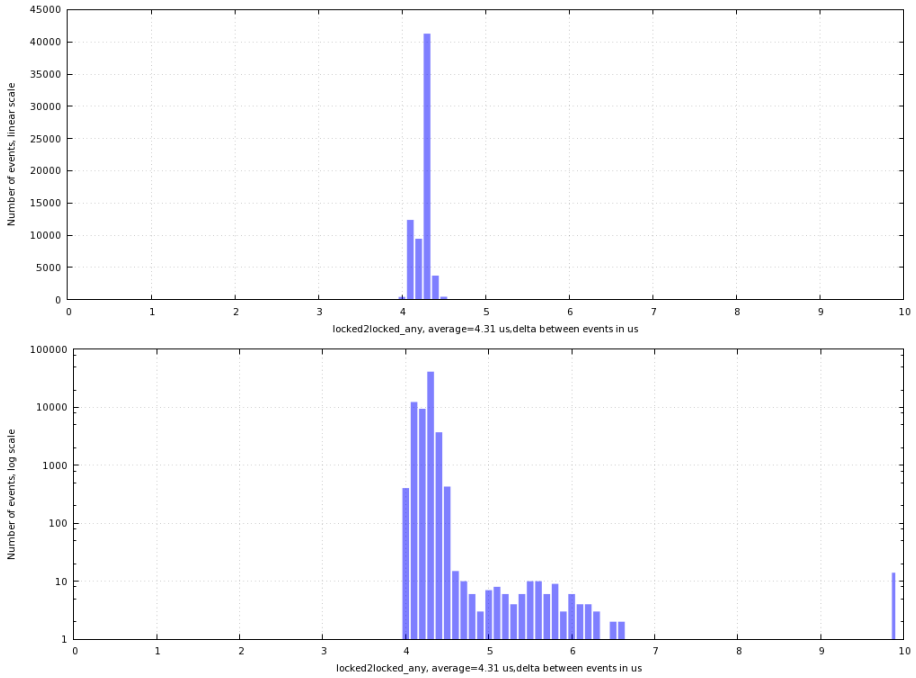

Figure 6. Distribution for Locked-to-Locked on z13. The Load-and-Test before the Compare-and-Swap dramatically improves the performance.

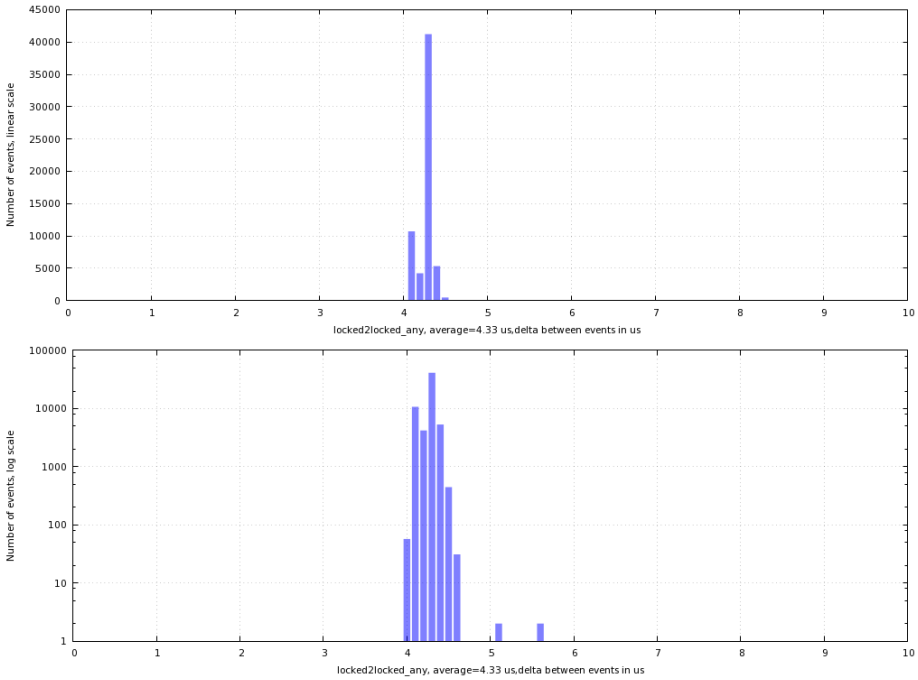

Figure 7. Distribution for Locked-to-Locked on z13. 32 CPUs are placed close to each other within the package (near placement). Therefore, the latency between CPUs is minimal. The critical section time is set to 4 micro seconds. The idle time has been set to a large value, so that there is no interference between iterations. A total of 2000 iterations was run. ‘near placement results in excellent performance.

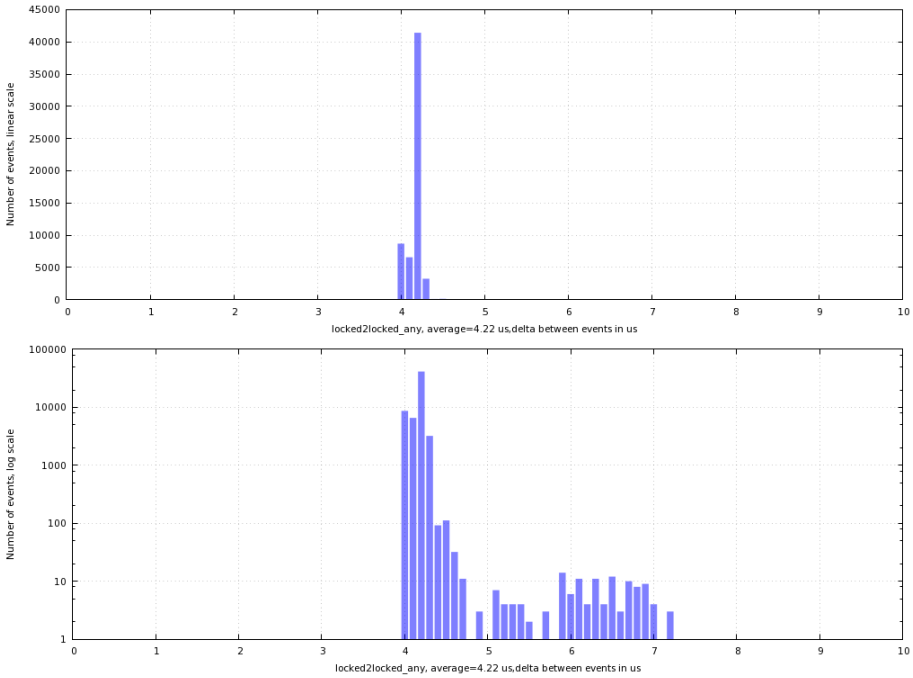

Figure 8. Distribution for Locked-to-Locked on z13. 32 CPUs are distributed across the entire package (distributed placement). Therefore, the latency between CPUs is larger than for ‘near placement’ and the routing is more complex. The critical section time is set to 4 micro seconds. The idle time has been set to a large value, so that there is no interference between iterations. A total of 2000 iterations was run. ‘distributed placement still results in good performance. However, it is less optimal than ‘near placement’.

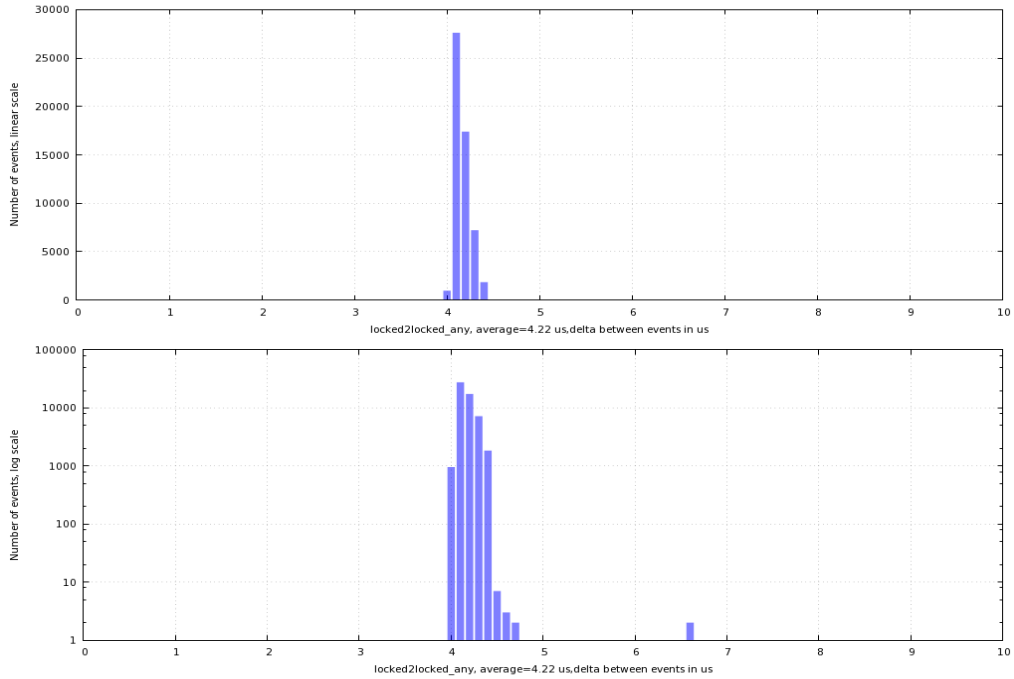

Figure 9. Data is from a run on z13 on 32 CPUS that are distributed by 'near placement' across the system. The critical section time is set to 4 micro seconds. The idle time has been set to a large value, so that there is no interference between iterations. A total of 2000 iterations was run. This data is based on no false sharing. When applying unintended read sharing the efficiency drops (see Figure 10).

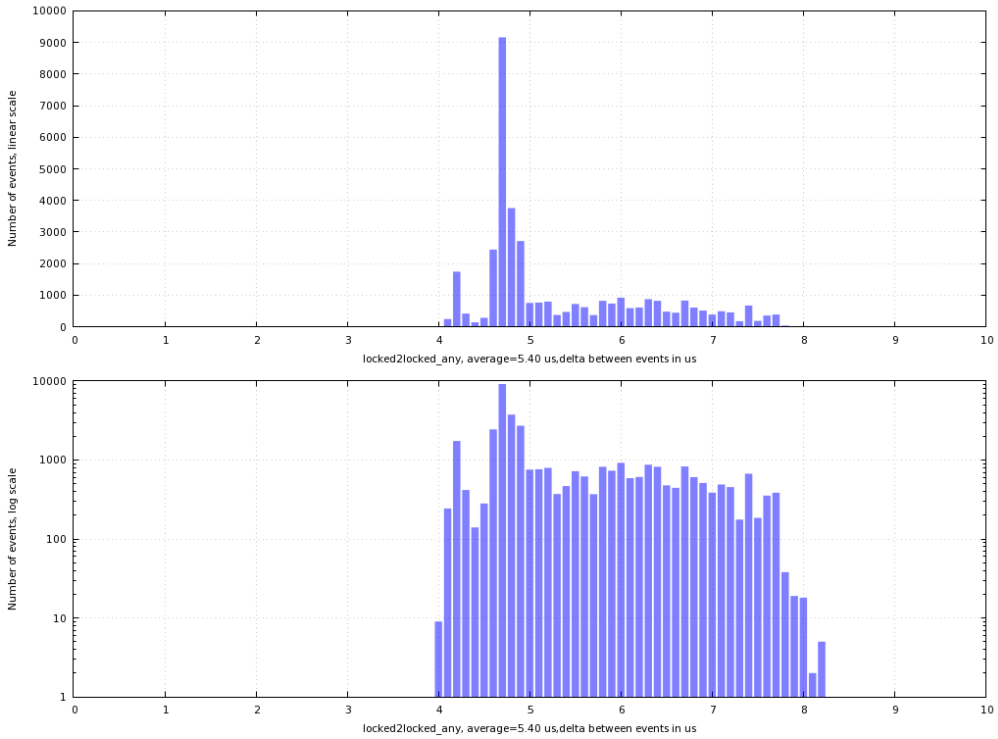

Figure 10. Data is from a run on z13 on 32 CPUS that are distributed by 'near placement' across the system. The critical section time is set to 4 micro seconds. The idle time has been set to a large value, so that there is no interference between iterations. A total of 2000 iterations was run. In contrast to Figure 9, extreme read sharing has been applied within the non-critical section. This results in a noticable performance degredation.