Temporal Decoupling –
Are “Fast” and “Correct” Mutually Exclusive?

Jakob Engblom, Intel, Stockholm, Sweden (*jakob.engblom@intel.com*)

***Abstract*— Temporal decoupling is a crucial virtual platform performance technique – without it, a virtual platform (VP) is destined to be unusably slow for large workloads. But temporal decoupling affects how the components of a VP interact and thus potentially alters the behavior of software running on the VP. Finding the “best” level of decoupling is thus a matter of balancing conflicting requirements. In this paper, we present our experience, measurements, and experiments with temporal decoupling across a wide range of software workloads and processor and hardware interactions, providing insight into the speed-behavior trade-off and the interesting things that can happen.**

Keywords—virtual platform, performance, software testing, tools,

I. Introduction

A virtual platform (VP) is a software program that simulates computer hardware, with the goal of running the same software as the hardware platform, see for example the various tools and platforms from [1][2][3][4]. Virtual platforms (VPs) are absolutely necessary in all SoC and platform development projects today. VPs enable pre-silicon development, testing, and validation of software, from low-level firmware and boot code to operating systems and application programs. VPs provide software developers with access to future hardware, and enable system development and integration before silicon is available. VPs execute automated tests for continuous integration, both for pre-silicon and post-silicon hardware, and run with field-programmable gate array (FPGA) prototypes and emulators in hybrid setups… the application areas are many and varied. For most software use cases, speed is critical. The faster the VP runs, the bigger and more realistic software loads can be executed – such as databases and database benchmarks on multi-socket server platforms [5]. In this paper, we focus on the application of VPs to software executions and test, and how to increase VP performance.

VP performance has increased over the years thanks to technologies such as functional modeling and transaction-level modeling [1][2][6], direct access to simulated memory from models (such as SystemC TLM-2 Direct Memory Interface [6] or the Wind River® Simics® Simulator Translation Cache (STC) [7]), fast interpreters [7], just-in-time (JIT) compilers from target code to host code [3], the use of virtualization technology to run target code [8][9], and optimizing the execution of idle systems. These techniques all reduce the amount of work needed to complete a single instruction or other step in the target system execution. Orthogonal to all these techniques is the use of temporal decoupling to improve simulator performance by increasing both temporal and spatial locality.

When using temporal decoupling, a part of the simulated system is allowed to run ahead in time before synchronizing with other parts of the system. This increases the locality in the simulator and reduces the amount of overhead code being executed compared to the amount of useful code. All other things being equal, temporal decoupling can provide for performance improvements of up to a thousand times compared to a naïve “switch on every instruction or cycle” schedule. However, temporal decoupling also changes the relative timing of various events in the simulator, and this can potentially create strange effects that might be observable from software. This paper will examine this aspect to some depth, based on fifteen years of practical experience running with increasingly aggressive temporal decoupling in the Simics virtual platform ecosystem [1][7], and talking to users from various communities about the use of temporal decoupling. The results can be considered qualitative and anecdotal, but there are some clear overall patterns that emerge.

We will discuss the effect of temporal decoupling in the context of single-threaded execution of a simulator. Temporal decoupling semantics are used almost universally to enable parallel virtual platform execution techniques [1][10][11][12][13], and the semantic effects are basically the same as for single-threaded execution. However, the performance effects are more variable due to parallel execution being inherently more noisy and sensitive to synchronization overheads.

II. Temporal Decoupling

Temporal decoupling has been in use for at least fifty years in computer simulation, coming into use almost as soon as we got multiprocessor systems. In 1967, Fuchi et al published a paper describing a simulator for a multiprocessor system [9]. In the paper, they describe the problem and technique of temporal decoupling:

```
The TSS [Time Sharing System] technique and the notion of "coroutine" can be used to
simulate this. Each coroutine corresponding to a CPU, device or channel is run with a
time-slice base. The smaller the time-slice quantum becomes, the more accurate the
simulation of the parallelism is.
```

A. Basic Definition

To discuss temporal decoupling in this paper, we will use a multiprocessor system with shared memory and a small set of peripheral devices as shown in Figure 1. We call each part of the platform a simulation object (we have P1, P2, P3, D1 to D5, and the memory and interconnect in the example). To simulate this with temporal decoupling, we assume that we have a given time quantum Q, which is the maximum amount of time that we let a time domain run ahead of other time domains in the simulation. The VP scheduler will run each separate time domain for a time quantum, and then switch to the next.

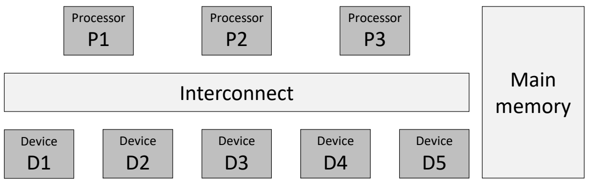


Figure 1. Simple example system

There are two common ways to group the simulation objects into time domains. In variant 1, most objects are active and have their own time domains and run in their own time quanta. In variant 2, only some objects (typically processors) run in their own time quanta, with other objects running when called from the active objects. In the author’s experience, variant 1 is common in SystemC-based virtual platforms, while variant 2 is common in virtual platforms that belong to the “computer architecture tradition” of SimOS [13], Simics [1], Qemu [3], and similar. Note that both variants consider main memory to be a shared passive resource that does not run in its own time domain, and that some systems mix the two.

B. Variant 1

In Variant 1, each object is considered active and runs in its own time domain, as shown in Figure 2 (the figure merges the interconnect and memory to better illustrate the way that all time domains directly change the memory contents). As shown in Figure 3, memory operations are visible immediately to all other objects. P2 sees the effects of the memory writes done by P1 at the start of its quantum – basically moving information backwards in time.

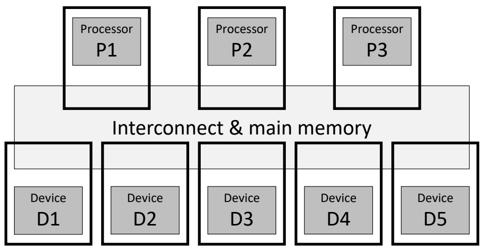


Figure 2. Variant 1 temporal decoupling platform split

Other interactions take longer to propagate. When processor P1 sends a request to device D1, D1 will process the request in its own time quantum, and P1 will see the result at the beginning of its next time quantum. The same holds when P1 sends a request to D3, and D3 in turn sends a request to processor P2 – it is visible at start of the second time quantum for P2. This means that actions like interrupts between processors and timer interrupts from devices to processors will appear only at the start of time quanta, and that propagation will take at least one time quantum. Depending on the implementation, it is possible that only the last action will be seen (with later actions

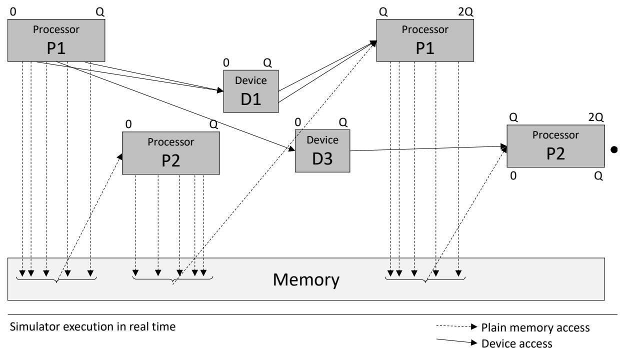


Figure 3. Variant 1 temporal decoupling, execution pattern and visibility of events between time domains

of the same type overwriting earlier actions), but ideally they should be buffered in such a way that all actions are made visible.

C. Variant 2

In Variant 2 temporal decoupling, devices run in the time domain of a processor, as shown in Figure 4. This design is based on the observation that devices don't usually take much time to simulate and that it is quite useful for the processor to directly interact with simple devices like timers and interrupt controllers without waiting for the end of the time quantum. Thus, devices end up being "slaved" to the processors in the simulator.

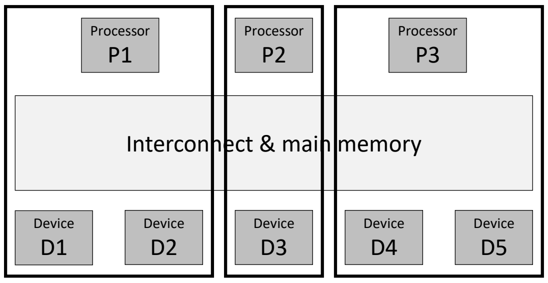


Figure 4. Variant 2 temporal decoupling time domain split

Figure 5 shows the same set of memory accesses and device accesses as seen in Figure 3, plus an additional path from P2 to D2. The execution pattern is different, since the devices are activated immediately when they are accessed by processors and do not run in their own time quanta. This changes the semantics of the platform, since the interaction between P1 and D1 is immediate, and the action chain from P1 to D3 to P2 is visible to P2 at the start of its first time quantum. When P2 accesses D2, however, that becomes visible to P1 in its next quantum – thus, the order of scheduling impacts the delay. Our experiments in this paper are performed using Simics, which is a Variant 2 virtual platform. The observations are also valid for Variant 1 virtual platforms.

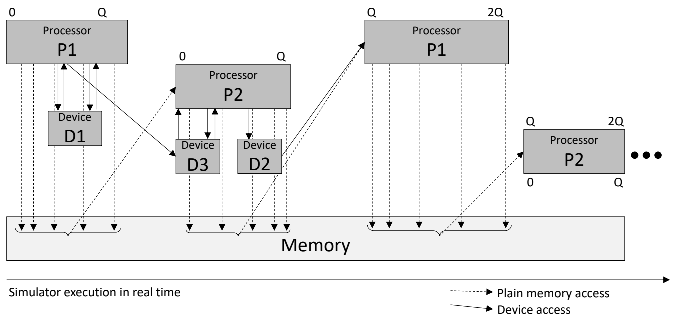


Figure 5. Variant 2 temporal decoupling, execution order and event visibility

D. Scheduling Precision

Temporal decoupling as described in this paper assumes a cooperative scheduling model, which is used in all current virtual platform frameworks that we are aware of. In such a framework, it is crucial that models hand back execution to the scheduler when their time quantum is over. Some models we have seen do not do this with perfect precision, which breaks platform repeatability and determinism – but it really does not affect the overall results of this paper.

E. Dynamic Time Quanta

Another approach is to dispense with a fixed time quantum, and instead have simulation objects pause their execution and switch to other objects when they communicate as described in [11] and [16].The published information on this approach focuses on the modeling of time inside of the simulation objects, rather than on how to run code on instruction-set simulators (ISS). If we compare this to variant 2, the likely effect is to cut the quantum on each device access, which will reduce the time quanta to the length of time between device accesses. In our experience, this time is usually far shorter than the desired time quanta of 10k to 1M cycles (see below). For software running on an ISS, it is also hard to determine communication points - since every memory access can in principle be a communications event.

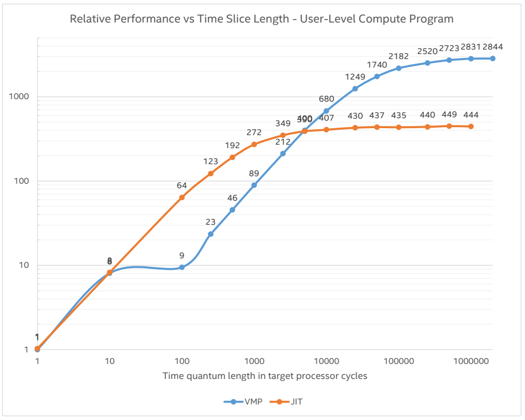
F. Real Concurrency

It is also possible to run a multicore target system without using round-robin scheduling and time quanta at all, rather relying on host threading to run each virtual processor as a thread. This is the approach taken by virtual machine systems like VMware*, Virtualbox*, and Linux* Kernel-based Virtual Machine (KVM), where device simulation is minimal and determinism and repeatability are not important. Such systems create virtual copies of the underlying host, and rely on the host hardware to handle synchronization – atomic instructions are executed directly on the host, and host memory is directly used by the processors and thus changes propagate between processors via the host's cache coherency.

III. Basic Performance Benefits

Before going into the details on the semantic effects of temporal decoupling, let's look at the raw performance benefit of temporal decoupling. In Figure 6, we show simulation performance, measured when simulating a four-core Intel Architecture target with a functional model of processors and memory. We ran both a compute-intense user-level program and a Linux boot. Each test takes a few hundred billion target instructions to complete. "VMP" is the Simics technology that uses Intel® Virtualization Technology for IA-32, Intel® 64 and Intel® Architecture (VT-x) to run Intel® architecture-based target code directly on the host. We can see that VMP benefits more from long time quanta, while the JIT reaches maximum performance around 10k instructions. The performance increase for user-level code going from single-cycle time quanta is more than 400x for JIT and 2800x for VMP. This happens since VMP executes code faster than JIT, but with a higher overhead for entering and exiting target code execution.

The speedup for any particular workload will vary with the nature of the software and the simulated system, but the overall pattern is consistent: longer quanta means faster execution. Some other examples include [2], in which the author with colleagues reported on the effect of time quanta on JIT execution of PowerPC* code, showing a performance increase of 100x when going from a time quantum of 10 cycles to 100k cycles. Jonack and Ambel show a significant performance increase for a DSP ISS when going from 1 to 1000 cycles time quanta [14], and the OVPSim* simulator defaults to 1ms as the time quantum length in order to optimize performance [15].



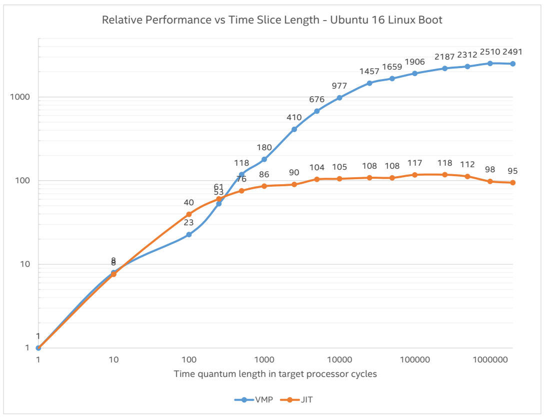


Figure 6. Performance benefits of temporal decoupling, for a user-level program and Linux boot on a four-core Intel target

IV. Race conditions

Race conditions are a common problem in multithreaded and concurrent software. Does temporal decoupling affect the ability to reproduce software race conditions in a virtual platform? Our experience is that it has a dual effect – long time quanta will make race conditions trigger less often (since there are fewer points where threads interact), but varying the time quantum length actually helps find races and to reproduce races from the real world.

We have explored the effect on this using a worst-case microbenchmark that has a number of threads doing read-modify-write on a shared memory location without any locking or synchronization. Using a time quantum of

10k cycles rather than 10 makes race error approximately 1000 times less common – but they *do* appear even with the longer time quantum. Race conditions in real software are typically much more intermittent, and to find them it is more important to run through a lot of code under varying conditions than to present a precise model of thread-to-thread interactions. Thus, long time quanta makes the use of the simulator more effective overall.

Since the length of the time quantum affects the synchronization and communication between different software threads and processors, varying the time quantum has proven to be a very useful tool to find race conditions and provoke latent bugs in software. For example, this has been used to debug Simics using Simics itself [18]. It is a simple and effective way to “shake the software” a bit to make bugs fall out with no need to change the model. Another technique that has been used is to change the relative speed of processors in a system –for example by changing the virtual clock frequency of a core, or the time that each instruction takes, thus doing more or less target work in each virtual time unit (which really has nothing much to do with temporal decoupling per se).

Note that when debugging and analyzing low-level multicore lock code such as found in OS kernels, it makes a lot of sense to have the time quantum set to one. That will show operations across processors step by step and cycle by cycle. One technique that has been used is to first run quickly with high time quanta to a point in time where locking is about to begin – and then dial down the time quantum before going through the lock code.

V. Fairness

Temporal decoupling might also affect the *fairness* between concurrent target software threads accessing a shared resource. Examples include shared variables, hardware devices accessed by multiple separate processor cores, or shared memory interfaces or cache hierarchies. In these cases, a software thread holds a shared resource for some time to the exclusion of others. If accesses to the shared resource are very frequent, we typically see some processors in the round-robin execution schedule having much more access to the resource since the time quanta makes it easier to “hog” the resource. The reason this happens is that if a processor ends its time quantum while it is holding the resource, none of the other processors will be able to use the resource in their time quanta.

This effect can be seen clearly in micro-benchmarks and simulating memory system performance. However, it is not necessary to dial back the time quantum to one cycle to mitigate the effect. It seems sufficient that the time quantum is shorter than the length of the blocking. For example, experiments on the Sulima simulator [10] indicate that errors in cache simulation are small when using temporal decoupling time slices less than the length of an L2 cache miss.

Figure 7 shows the result of an experiment where a program with four threads is run on a simulated eight core machine running Linux. Each thread contains a loop that first has a short lock section where shared variables are accessed, and then a wait section of varying length. We did four runs of the program for each of a series of wait section lengths. With long time quanta, we see that some threads acquire the lock significantly more often than other threads - the “unbalance” measure in Figure 7 indicates the difference between the thread with the most and

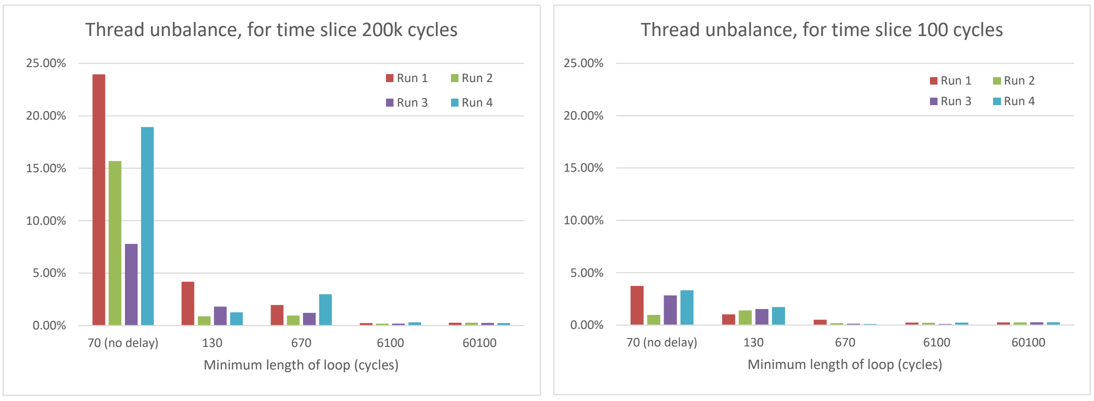


Figure 7. Measuring the unbalance in threaded software caused by temporal decoupling

the least lock acquisitions. However, as the wait section becomes larger, the effect rapidly diminishes. Thus, we can conclude that temporal decoupling does have an impact on fairness, but only when the software is using shared resources very frequently. It should be noted that running the experiments with a time quantum of 100 cycles took about 100x longer than running with a time quantum of 200k cycles.

VI. Timestamps and Synch Algorithms

Software that observe time stamps across processor cores (and hardware devices) and might get into problems when run on a platform with temporal decoupling. In our experiments and experience, we work with a model and all time readings are based on the local time of the active processor, and not on a global kernel time.

We came across this phenomenon in the early days of multicore computing (around 2005), when the Linux kernel for PowerPC* featured an algorithm to synch up the cores that was sensitive to temporal decoupling. The effect is shown in Figure 8, where we can see that as the time quantum gets longer, the total number of instructions executed and thus the virtual execution time goes up – slowly at first, and then rather dramatically. On the other hand, the real-world execution time is the lowest around 10k cycles thanks to the performance benefits of temporal decoupling.

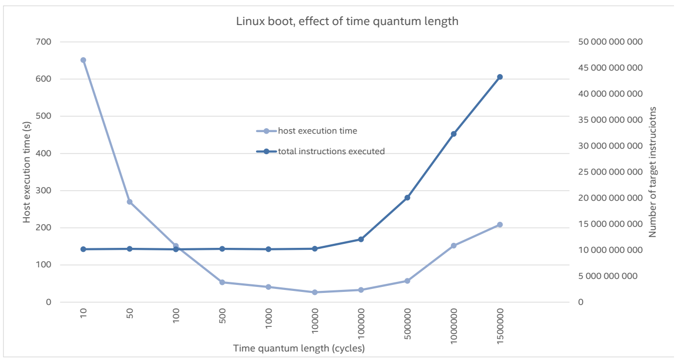


Figure 8. Temporal decoupling effects on an early Linux multicore boot

Another problem we have seen is when software reads a timer value from a shared timer and writes it to memory where another core can see it. If the other core then reads the same timer, what should be reported from the hardware model? If the timer reports the local time of the processor core doing the read, time might appear to be going backwards depending on where in the time quantum the read happens. This is often harmless, but we have seen software that expects time to monotonically increase – which requires playing some games with time in the timer device in order to allow long time slices. For example, remembering the highest reported time so far in a time quantum, and making sure never to report a smaller one. Or, having the timer be aware of time quanta, and showing each processor in order a separate set of time stamps (the first processor of *N* sees 0 to *Q/N*, the next sees *Q/N* to *Q/N*2*, etc.) Similar timing tricks are reported for FIFOs in [16].

Finally, there are cases where software just won't work with long time quanta, and will stubbornly fail no matter what. In that case, dialing down the time quantum and suffering low performance might be the only choice.

VII. Ping-Pong Protocols

"Ping-pong" protocols, where a message is sent from one unit, processed in another unit, and then a reply is sent back to first unit can be affected by temporal decoupling. If the units are in separate time domains, each message takes at least a time quantum to propagate, making the round-trip time at least two time quanta. When the time quantum is large, this can make software progress very slow, even if the simulator itself is running very quickly in terms of getting through virtual time.

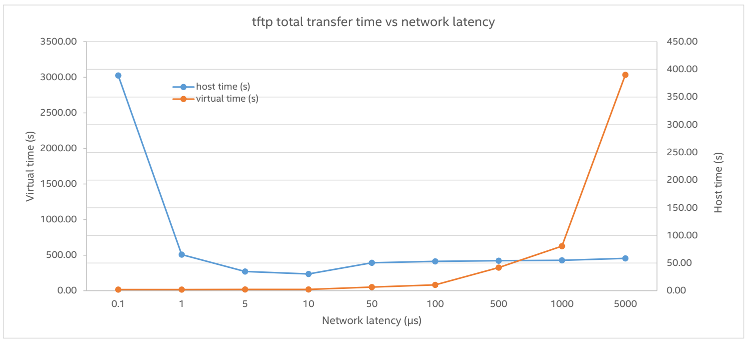


Figure 9. tftp transfer with varying network latencies

This effect has been observed when simulating networks. While long time quanta typically work fine with protocols like TCP that are built to handle the latency of the Internet, there are protocols that are sensitive to time quantum lengths – such as tftp, as shown in Figure 9. The test copies a file from one simulated machine to another using tftp, with varying network latencies (time quantum lengths).

Figure 9 is reminiscent of Figure 8 – but with the difference that even as the virtual time increases dramatically, the simulation run time on the host stays mostly the same. This is thanks to the effect of idle time optimization (waiting for a network packet is done by OS idling which can be detected and optimized). Thus, regardless whether the transfer takes 300 or 3000 virtual seconds, it always takes around 60s to execute on the host. At the highest latency, the simulator is running through 100 billion target cycles per host second, or roughly 50x real-time speed.

When two processors communicate over shared memory or another shared resource, we are rarely as lucky as in the network case that the wait for the other side is done using efficient idling. Instead, polling is common, and this can slow down the simulation enormously as a processor basically spends its entire time quantum reading a value that will not change. This case is sometimes handled by "breaking the quantum", where the processor doing the reading immediately finishes its quantum and lets the other objects run in order to produce a reply as quickly as possibly [11]. Another way to handle polling across time domains is to stick to the time quanta – but make the read operations take a long time. In this way, rather than doing thousands of reads during its quantum, the processor might make only a handful. This technique is often used in Simics models to tweak performance.

VIII. Multicore Scaling Evaluation

Finally, let's take a look at how time quantum lengths affect software scaling studies. Even without detailed hardware performance models, running parallel software on a virtual platform will expose software-internal scaling problems like poor synchronization schemes – in particular when considering the use of OS APIs.

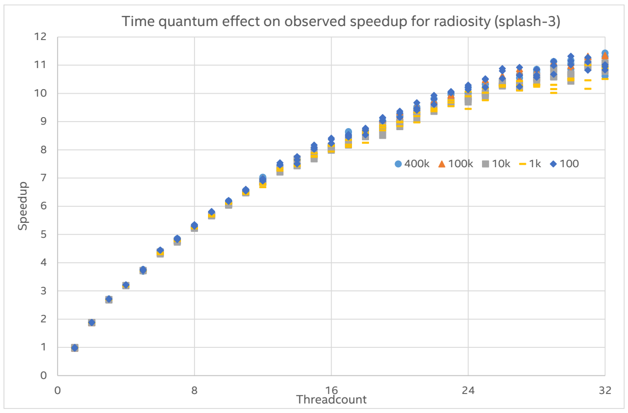


Figure 10. Speedup for 1 to 32 target threads, for different time quanta

Figure 10 shows the results of running the Radiosity program from the Splash-3 benchmark suite [17] on a 32-core Intel Architecture target running Ubuntu Linux 16.04. The benchmark is run with one to 32 compute threads, and with the time quantum varied from 100 to 400k cycles. What we see is that regardless of the time quantum, overall we see the same scaling effects. Higher time quanta appear to give slightly higher scaling numbers, but not to the extent that the results are misleading. This effect is likely explained by the fact that the software is not synchronizing all that often compared to the length of the time quanta, and thus the effect of long time quanta affecting locking patterns is minimal. This is similar to what was discussed in the section on fairness above.

IX. Summary

Temporal decoupling is a crucial technique for increasing the performance of virtual platforms. Time quantum lengths of 10k to 1M cycles are needed to maximize VP performance. Most of the time, software functionality and correctness are unaffected by temporal decoupling, and the default should be to use long time quanta. However, sometimes it is necessary to reduce the time quanta in order to observe certain software or hardware effects, or satisfy software that is very particular about synchronization across cores.

Acknowledgments

I would like to thank the anonymous paper reviewers who provided good input on the kinds of questions that made sense to address in the final version of the paper. I also thank the members of the Simics team.

REFERENCES

[1] Daniel Aarno and Jakob Engblom, Software and System Development using Virtual Platforms – Full System Simulation with Wind River Simics, Morgan Kaufmann Publishers, 2014.

[2] Rainer Leupers and Oliver Temam, (Editors). 2010. *Processor and System-on-Chip Simulation. Springer*. ISBN 978-1-4419-6175-4, 2010.

[3] Fabrice Bellard, "Qemu, a fast and portable dynamic translator", *Proceedings of the 2005 USENIX Annual Technical Conference*, Vol. 41. 2005.

[4] S. Koerner, C. Werner, T. Hess, P. Schulz, M. Strasser, S. Wagner, H. Böhm, M. Troester, D. Bolte, H. Elfering, T. Pohl, K. Theurich, W. H. Miller, and P. Szwed. "Firmware verification and simulation in IBM zEnterprise 196", IBM Journal of Research and Development, Volume 56, Issue 1/2, January-March 2012.

[5] Jakob Engblom, *Running "Large" Software on Wind River® Simics® Virtual Platforms, Then and Now*, Intel Developer Zone blog post, 15 March 2018. URL: https://software.intel.com/en-us/blogs/2018/03/15/software-on-wind-river-simics-virtual-platforms-then-and-now

[6] IEEE Std 1666-2011, IEEE Standard for Standard SystemC® Language Reference Manual, IEEE, January 2012.

[7] Magnusson, P., Dahlgren, F., Grahn, F., Karlsson, M., Larsson, F., Lundholm, F., Moestedt, A., Nilsson, J., Stenström, P., Werner, B. "SimICS/sun4m: A Virtual Workstation", *Proceedings of the 1998 USENIX Annual Technical Conference*, June 1998.

[8] Andreas Sandberg, Nikos Nikoleris, Trevor E. Carlson, Erik Hagersten, Stefanos Kaxiras, David Black-Schaffer: "Full Speed Ahead: Detailed Architectural Simulation at Near-Native Speed", *Proceedings of the 2015 IEEE International Symposium on Workload Characterization (IISWC)*, Atlanta, Georgia, USA, October 2015.

[9] Fuchi, K., Tanaka, H., Manago, Y., Yuba, T. "A program simulator by partial interpretation," *SOSP '69: Proceedings of the second symposium on Operating systems principles, pp. 97–104*, October 1969.

[10] A. Over, B. Clarke and P. Strazdins, "A Comparison of Two Approaches to Parallel Simulation of Multiprocessors," *2007 IEEE International Symposium on Performance Analysis of Systems & Software*, pp. 12-22, San Jose, CA, 2007.

[11] Denis Becker, Matthieu Moy, and Jérôme Cornet: "Challenges for the Parallelization of Loosely Timed SystemC Programs", *Proc. 2015 International Symposium on Rapid System Prototyping (RSP)*, October 2015.

[12] N. Manjikian, "Parallel simulation of multiprocessor execution: Implementation and results of SimpleScalar," *Proceedings of the 2001 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Nov 2001, pp. 147–15

[13] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod, "Using the SimOS machine simulator to study complex computer systems," *ACM Transactions on Modeling and Computer Simulation*, vol. 7, no. 1, pp. 78–103, Jan 1997.

[14] Rocco Jonack and Juan Lara Ambel. "VP Performance Optimization – How to Analyze and Optimize the Speed of SystemC Models", *Proceedings of the Design and Verification Conference Europe (DVCON Europe)*, München, 14-15 October 2014.

[15] Imperas Software Limited, *Advanced Simulation Control of Platforms and Modules User Guide v 2.0.1*, October 2017

[16] Claude Helmstetter, Jérôme Cornet, Bruno Galiléey, Matthieu Moy, and Pascal Vivet, "Fast and Accurate TLM Simulations using Temporal Decoupling for FIFO-based Communications", *Design, Automation, and Test in Europe (DATE)*, March 2013.

[17] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research", *Proceedings of the 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2016.

[18] Jakob Engblom, *Debugging Simics on Simics*, Wind River blog post, 5 December 2017. URL: http://blogs.windriver.com/tools/2012/12/debugging-simics-on-simics.html

NOTICE AND DISCLAIMER