Accelerating Graphics in the Simics Full-system Simulator

Eric Nilsson*, Daniel Aarno†, and Erik Carstensen‡
Software and Services Group
Intel Corporation
Stockholm, Sweden
*Email: {*eric.nilsson, †daniel.aarno, ‡erik.carstensen}@intel.com*

Håkan Grahn
Department of Computer Science and Engineering
Blekinge Institute of Technology
Karlskrona, Sweden
Email: hakan.grahn@bth.se

***Abstract*—Virtual platforms provide benefits to developers in terms of a more rapid development cycle since development may begin before next-generation hardware is available. However, there is a distinct lack of graphics virtualization in industry-grade virtual platforms, leading to performance issues that may reduce the benefits virtual platforms otherwise have over execution on actual hardware.**

This paper demonstrates graphics acceleration by the means of paravirtualizing OpenGL ES in the Wind River Simics full-system simulator. We propose a solution for paravirtualized graphics using magic instructions to share memory between target and host systems, and present an implementation utilizing this method. The study illustrates the benefits and drawbacks of paravirtualized graphics acceleration and presents a performance analysis of strengths and weaknesses compared to software rasterization. Additionally, benchmarks are devised to stress key aspects in the solution, such as communication latency and computationally intensive applications.

We assess paravirtualization as a viable method to accelerate graphics in system simulators; this reduces frame times up to 34 times compared to that of software rasterization. Furthermore, magic instructions are identified as the primary bottleneck of communication latency in the implementation.

***Keywords*-simics; full-system simulation; paravirtualization;**

I. Introduction

Virtual platforms are becoming an important tool in the software industry in order to provide cost-effective time-to-market gains and meet the ever-shortening product life-cycles [1], [2], [3], [4]. Virtual platforms deliver these time-to-market benefits by enabling pre-silicon development [1], [4] and by providing tools such as deterministic execution, checkpointing, and reverse execution [4], [5]. These features are useful for debugging and testing a diverse range of software, from firmware to end-user applications [3].

There are several techniques to provide fast and functional virtual platforms that are running CPU workloads. Typical methods include interpretation [6], [7], just-in-time compilation [5], [7], and hardware-assisted virtualization [5], [3]. Virtual platforms using these techniques can typically achieve a simulation performance in the range of 10-1000 million instructions per second [5].

The GPU is a vital part in delivering good user experiences on many devices, ranging from wearable, hand held, and portable units, to desktop computers. The widespread use of GPUs and the increasing complexity of these electronic systems extend the virtualization needs of such devices. However, due to large architectural differences, delegating GPU workloads to CPUs may yield poor performance.

Instead, by neglecting some hardware compatibility, one may circumvent the virtual machine and delegate GPU workloads to the GPU of the simulation host. This way, host hardware can be utilized in a process known as ”Paravirtualization” [8]. Paravirtualization has been used to accelerate graphics in the past; most notably, Lagar-Cavilla et al. accelerate OpenGL 1.5 up to two orders of magnitude for WMware Workstation and Xen VMMs [9]. To relieve communication bottlenecks, Lagar-Cavilla et al. suggests using a shared memory model for target-to-host communications [9].

This paper presents the acceleration of OpenGL ES 2.0 in the Simics full-system simulator, using magic instructions to share VM memory directly from a simulated RAM image. The implementation is evaluated using performance benchmarks stressing important attributes of the devised solution, and subsequently compared to software rasterization on the simulated platform. Furthermore, the study identifies performance bottlenecks that may obstruct paravirtualized real-time graphics. The results presented in this paper show performance improvements of up to 34 times compared to software rasterized counterparts.

II. Methods and Results

OpenGL paravirtualization in Simics encompasses three overall components: the target system libraries, the host system libraries, and a communications channel between them named the ”Simics pipe”. Most OpenGL glue code, on both target and host, is generated by a program from specification files detailing function signatures and arguments. An exception is methods that require state saving, which are implemented manually.

The target system libraries implement the OpenGL and EGL (the interface to the underlying platform windowing system) APIs; unmodified binaries in the target system are linked with these libraries as usual. However, instead of

communicating with the graphics device, the target system libraries serialize and forward the command stream to the simulation host. The transmission is not necessarily performed at once, nor in the designated order, because of uncertainties regarding argument data proportion. For instance, the number of vertices to be rendered does not have to be apparent at a given time, but implicit in a later OpenGL invocation. Accordingly, certain paravirtualized function calls have to be delayed until more information is known about the OpenGL state.

In collaboration with the target system libraries, the host system libraries decode and interpret the received byte stream. Subsequently, the host system libraries may safely perform the relayed workload and return any results to the target system.

Both target and host system libraries maintain a subset of the OpenGL state, such as bound vertex buffers and attribute properties. These states must be maintained because of the asynchronous nature of the command stream.

Because of differences in the creation and maintenance of windows on different platforms (Fedora, Android, etc.), the window to which the target OpenGL application renders is kept on the simulation host. This is problematic; the target system libraries must communicate with a fraudulent window in the simulation host – *and* the native target window. For example, it is important that the native window reports successful initialization, lest the OpenGL application concludes an error and quits. The issue is overcome by selectively overriding symbols in the target libraries so that a subset of functions may be overloaded. This way, one may extend the original EGL library to invoke the simulation host prior to performing its actions.

To communicate with the simulation host, the Simics pipe uses ”magic instructions”. A magic instruction is a `nop` instruction that invokes a callback-method in the simulation host when executed on simulated hardware [3]. Because of the inherent performance demands brought on by real-time graphics, they constitute a suitable communications medium for rendering information between target and host systems.

During a magic instruction, we may utilize any available registers; the number and size of registers is the data-sharing bottleneck of this method. Thus, we transmit the starting address of the serialized command stream in a 64-bit register. Having escaped the simulation context, Simics can translate the transmitted virtual address to a physical one using the virtual machine MMU. Consequently, the physical address can be used to locate the memory page in the simulated RAM image. The pages constituting the buffer are locked in RAM, protecting the buffer from being paged to the swap area. This ensures that pages are not swapped to disk when the simulation state is paused. Subsequently, all memory pages are continuously retrieved by iterating the original virtual address with the target page size, effectively traversing the virtual memory table.

A. Experimental Methodology

To evaluate the implementation, performance of paravirtualized graphics in Simics is compared to software rasterization. Simics itself simulates an Intel® Core™ i7 processor and an Intel® X58 chipset. Throughout simulation, hardware-assisted virtualization using KVM runs x86 instructions natively on the host hardware. Like the host system, the simulation target runs Fedora 19 Linux and use the Mesa llvmpipe driver software rasterizer [10].

The experiments are performed on a system with the following specifications:

- Intel® Core™ i7-4770HQ
- Intel® Iris™ Pro Graphics 5200

Two benchmarks are devised on-site to stress suspected bottlenecks: one benchmark performs a large number of OpenGL invocations, while the other has a computationally intensive workload. Given a target frame time of $16\ \mathrm{ms}$, the benchmarks are configured to run at 10 to $20\ \mathrm{ms}$ per frame when hardware accelerated on the host system; a $16\ \mathrm{ms}$ frame time roughly corresponds to 60 frames per second. The benchmarks are shaped this way to reflect the expected load of a real-time interactive application. As such, the benchmarks should be representative of typical scenarios induced by modern applications using OpenGL, such as responsive UIs. The benchmark suite is open source [11].

For each benchmark, the elapsed times of 1000 frames are collected. To gain some understanding on how well the given performance scales, three instances of each benchmark are run with smaller and larger input data, tuned to yield approximately half and double frame time. The specifics of each benchmark are described below.

Benchmark: Chess: To stress the latency between target and host systems, the ’Chess’ benchmark performs a multitude of lightweight OpenGL invocations per frame, rendering a grid of chess-like tiles. For each frame rendered, depending on the number of tiles, the benchmark performs a large number of magic instructions. This induces high utilization of the Simics Pipe, which is intended to stress suspected magic instruction overhead.

A long sequence of draw calls is representative of drawing multitudes of shapes with OpenGL, such as a UI. Accordingly, the benchmark is suitable for the purpose of representing a large number of graphics invocations.

The Chess benchmark is run with 60×60, 84×84, and 118×118 tiles. In every frame, 9 magic instructions are performed for each tile.

Benchmark: Julia: To stress the computational prowess of paravirtualized graphics in Simics, the ’Julia’ benchmark performs a lone, computationally intensive, OpenGL invocation that renders the Julia fractal [12]. The load of a fractal computation can easily be tuned by adjusting the number of iterations per pixel. Therefore, the Julia benchmark is suitable to profile a computationally intensive workload.

Table I. SOFTWARE RASTERIZATION RESULTS IN SIMICS.

Benchmark	Input	Elapsed time (ms)			
		Min	Max	Std	Avg
Chess	60×60 tiles	76.88	439.27	19.48	114.41
	84×84 tiles	167.36	402.87	9.38	192.08
	118×118 tiles	238.86	701.16	17.63	259.54
Julia	225 iterations	397.82	2183.99	83.45	461.64
	450 iterations	744.91	2662.67	62.88	776.41
	900 iterations	1338.40	2669.19	113.87	1415.23

Table II. PARAVIRTUALIZATION RESULTS IN SIMICS.

Benchmark	Input	Elapsed time (ms)			
		Min	Max	Std	Avg
Chess	60×60 tiles	185.19	362.23	11.19	201.85
	84×84 tiles	309.08	401.26	15.51	336.83
	118×118 tiles	612.77	719.14	14.12	650.67
Julia	225 iterations	21.89	47.65	3.82	26.56
	450 iterations	38.79	101.56	10.01	49.83
	900 iterations	34.74	156.35	8.55	41.81

The Julia benchmark is run with 225, 450, and 900 iterations, all of which induce 16 magic instructions per frame.

B. Threats to Validity

Because of complications caused by virtual time, measuring time in system simulation sometimes dictate special measures. For instance, in terms of real-time rendering, the observer is far more interested in a frame rate relative to wall-clock time rather than virtual time.

In order to measure frame time in relation to wall-clock time, profiling must take place outside of the simulation. One way of achieving this is to listen in on activity passing through a target serial port; this is a traditional front-end to the machine. In this way, a simulation breakpoint can be triggered at the occurrence of a certain sequence of bytes written to a UART serial port. This is the method used to measure frame time in Simics.

When using serial ports in this manner, one may introduce a profiling cost. For example, file descriptors do not immediately transmit a byte sequence over the system UART. For our set-up, we measure this overhead cost to be, on average, 1.5 ms. This average is subtracted from presented measurements.

C. Results

Results accumulated from software rasterized and paravirtualized execution are presented in Tables I and II. In Fig. 1, the results are presented as histograms, visualizing elapsed time in milliseconds to sample density. For each experiment, collected frame time samples (1000) are subdivided into 100 bins. Any measurements outside of the standard deviation are not included in the figures.

Fig. 1 indicates that the Chess benchmark, when software rasterized, yields a broad sample distribution, seemingly distributed around a single point. The right-hand side of the graph, showing impaired performance induced by paravirtualization, visualize a distribution decrease. This is supported by the data presented in Table II. We observe that software rasterization outperforms its paravirtualized counterpart, regardless of the number of tiles rendered. The benchmark is devised to identify any bottlenecks related to the number of paravirtualized function calls. Evidently, the prediction of a target-to-host communication latency issue is confirmed.

In Simics, magic instructions incur a context switch cost when resuming execution on the host. This causes the simulation to no longer execute natively, which inhibits performance improvements granted by hardware-assisted virtualization. It also entails that Simics can no longer utilize JIT, forcing the simulator to rely on interpretation. This explains why magic instructions have such a great impact on performance.

By measuring the elapsed time of 1000 magic instructions, we conclude that this induces an average overhead of 5 ms (not taking into account any profiling cost). These findings indicate that magic instruction overhead could account for the majority of elapsed frame time.

Fig. 1 shows that the Julia benchmark yields double to triple peak sample density distribution, both in software rasterized and paravirtualized Simics. What causes this behavior is unclear; the fractal algorithm should perform roughly equally frame-to-frame. The benchmark is intended to demonstrate how paravirtualization performs under computational stress, which is where the benefits of hardware acceleration should be made apparent. Accordingly, weaknesses in software rasterization is highlighted with frame times above the two second mark; corresponding paravirtualized maximum frame time measures a mere 156 ms. This confirms performance improvements of using paravirtualization with magic instructions to accelerate graphics.

These measurements are collected using hardware-assisted virtualization for accelerated performance. If hardware-assisted virtualization is not available, such as if the simulated platform is other than x86, we expect a major hit to performance. For software rasterization, this impact accounts for frame time increases well over two orders of magnitude. Meanwhile, performance impacts to paravirtualized performance is often not significant, sometimes as low as one third of the original frame time. For the Chess benchmark, paravirtualized frame time increases with *up to* one order of magnitude, one order less than the performance hit to software rasterization. Across the board, paravirtualization suffer less performance impact, rendering the benchmarks up to three orders of magnitude faster than software rasterization. We conclude that the effects of paravirtualization increase by one order of magnitude without hardware-assisted virtualization. This entails that workloads that are otherwise sub-optimal for paravirtual-

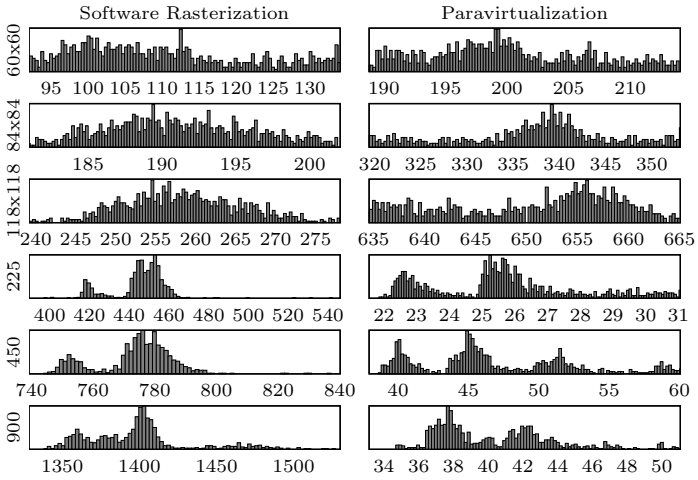

Figure 1. Sample histograms depicting 1000 frames subdivided into 100 bins, presented in milliseconds. Top 2×3: Chess. Bottom 2×3: Julia.

ization – those performing a large amount of function invocations – bring about performance improvements. Thus, the impact of magic instruction overhead is reduced, likely because a costly context switch is not inflicted on Simics. We reason that some software rasterized workloads (Chess) may attain decent simulation performance simply because of a fast simulator; when native execution is not available, neither JIT nor interpretation may attain the same speeds as paravirtualization.

The samples collected without hardware-assisted virtualization are not presented in detail in this paper.

III. Conclusion

This paper presents graphics acceleration by the means of paravirtualization in the Simics full-system simulator. The implementation generates EGL and OpenGL libraries and communicates with low-latency magic instructions. To evaluate the implementation, benchmarks are developed to highlight solution weaknesses and strengths; an analysis of results is presented, as well as benefits and drawbacks of paravirtualized methodology.

In Section II-C, compiled results showcase great improvements for computationally intensive graphics. Performance is improved by up to 34 times, reducing frame time from $1415\ \mathrm{ms}$ to $42\ \mathrm{ms}$; a frame time of $42\ \mathrm{ms}$ roughly corresponds to 24 FPS. Furthermore, paravirtualization lower maximum frame times, significantly improving standard deviation. Compared to software rasterization, paravirtualization is estimated to obtain an additional order of magnitude faster frame time without hardware-assisted virtualization.

Along with performance improvements, Section II-C detail a communications latency issue inherent in magic instruction overhead. This emerges as a performance bottleneck in great numbers of paravirtualized function invocations. While magic instructions are – evidently – fast enough to accommodate real-time graphics, improvements to this overhead should greatly improve their capacity.

To conclude: this paper demonstrates accelerated graphics in Simics using paravirtualization with magic instructions as a communications bridge. Graphics acceleration in Simics is relevant because it facilitates debugging, testing, and profiling of software that depends on GPU utilization. Consequently, paravirtualization is practical because it offers a good trade-off between development cost and performance. The findings of this paper may help in extending the use of Simics to application development dependent on GPUs, including computer graphics and general-purpose workloads.

The benefits of paravirtualized graphics are performance improvements of up to two orders of magnitude, paired with larger benefits in non-hardware accelerated use-cases. Magic instruction overhead is identified as the main performance bottleneck, resulting in a weakness to large amounts of framework invocations. Accordingly, the findings of this paper contribute to our understanding of the difficulties facing graphics acceleration in virtual platforms by demonstrating the use of paravirtualization as a successful formula to accelerate graphics in Simics.

References

[1] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.

[2] J. J. Yi and D. J. Lilja, "Simulation of computer architectures: Simulators, benchmarks, methodologies, and recommendations," *IEEE Trans. Comput.*, vol. 55, no. 3, pp. 268–280, Mar. 2006.

[3] R. Leupers and O. Temam, *Processor and System-on-Chip Simulation*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[4] D. Aarno and J. Engblom, *Software and System Development using Virtual Platforms: Full-System Simulation with Wind River Simics*.

[5] ——, "Simics* overview," *Intel Technology Journal*, vol. 17, no. 2, pp. 8–31, Dec. 2013.

[6] J. E. Smith and R. Nair, "The architecture of virtual machines," *Computer*, vol. 38, no. 5, pp. 32–38, May 2005.

[7] P. S. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenström, and B. Werner, "Simics/sun4m: A virtual workstation," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '98, 1998, pp. 10–10.

[8] L. Youseff, R. Wolski, B. Gorda, and C. Krintz, "Paravirtualization for hpc systems," in *Proceedings of the 2006 International Conference on Frontiers of High Performance Computing and Networking*, ser. ISPA'06, 2006, pp. 474–486.

[9] H. A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. de Lara, "Vmm-independent graphics acceleration," in *Proceedings of the 3rd International Conference on Virtual Execution Environments*, ser. VEE '07, 2007, pp. 33–43.

[10] Mesa Project, "llvmpipe," Available: http://bit.ly/19G8Xbp, accessed: 21-04-2015.

[11] Intel, "Opengl es 2.0 benchmarks," Available: http://bit.ly/IntelOpenGLES, accessed: 14-04-2015.

[12] J. Tsiombikas, "Fast and easy high resolution fractals with a pixel shader," Available: http://bit.ly/1xnKMcl, accessed: 21-04-2015.