Turning x86 into a Hardware Simulator for Future Manycores

Nils Asmussen Hermann Härtig Marcus Völp
Technische Universität Dresden
Nöthnitzer Straße 46
01187 Dresden, Germany
{asmussen, haertig, voelp}@os.inf.tu-dresden.de

ABSTRACT

Hardware/operating system co-design opens up many possibilities to experiment with new features and to propose new acceleration possibilities for evermore demanding applications. However, to justify these modifications, rigorous simulations and experiments are required. In this paper, we introduce a novel way of using high-end manycore systems as a simulation platform. Reserving some cores as application cores while using the others to simulate the new hardware features, we maintain near native performance and thus provide an early evaluation environment for application-level benchmarks. Illustrating our simulator with an envisaged inter-process communication mechanism for redirecting system calls to dedicated OS cores, we achieve a file-system throughput in a GPGPU-like setting that is close to the local case of more traditional settings where system calls invoke the local OS instance.

Categories and Subject Descriptors

C.1.3 [**Other Architecture Styles**]: Heterogeneous (hybrid) systems; D.4.4 [**Communications Management**]: Message sending

Keywords

manycore, simulator, operating system, minimal hardware

1. INTRODUCTION

Accelerator architectures such as GPGPUs, IBM Cell [1], and signal processors achieve their performance and energy advantages by offering a vast number of very simple and specialized cores. The tremendous core count of these architectures stems in part from a sacrifice of even the most fundamental hardware features that interacting applications and operating system kernels require. Our ultimate goal is to find new alternatives for providing these interaction possibilities without at the same time reintroducing the hardware complexity that we find in high-end manycore systems.

Our approach to reach that goal is a hardware/operating system co-design which allows us to analyze and evaluate the interplay of hardware and operating system in an iterative way. This requires a flexible implementation of the hardware to support quick and often changes. Additionally, in order to justify hardware modifications, a thorough evaluation is required, both in terms of microbenchmarks and ideally also in terms of data and compute intensive application-level benchmarks. In other words, we have to simulate new hardware features both in a flexible way and also at an execution speed that allows long-running application-level benchmarks.

Several simulators are described in the literature [5, 3, 13, 4]. However, common to all is a certain trade-off between accuracy of the simulated system and performance of the applications they run. As an initial step towards our goal, we decided to give up most of the accuracy in favor of speed and flexibility. That is, we do not intend to replace instruction level or cycle accurate simulators, which provide a much finer resolution for the simulated hardware at the cost of slow downs by approximately two orders of magnitude [13]. Instead, we would like to establish a platform and simulator to ease the hardware/operating system co-design and allow early experiments and evaluations of the interfaces of new components.

This is achieved by running Linux on an x86 manycore and using Linux mechanisms to simulate the hardware. Thereby we utilize the "even" cores for simulating new hardware components while preserving the "odd" ones for the applications. The rich programming environment of Linux allows us both an easy development and also rapid changes to the simulated hardware, while the dedicated cores for the applications enable us to achieve near native performance.

In this paper we describe the design and implementation of a first prototype of our simulation environment. The prototype realizes an experimental DMA interface for communicating between otherwise decoupled cores. As a transport medium we exploit shared buffers and the cache coherence protocol. Based on this DMA interface, we have implemented an inter-process communication (IPC) mechanism that allows applications to communicate with each other in an easy fashion. As a first benchmark of our simulator, we have used this IPC mechanism to redirect selected system calls from application cores and to a different core that executes the requested operating system functionality.

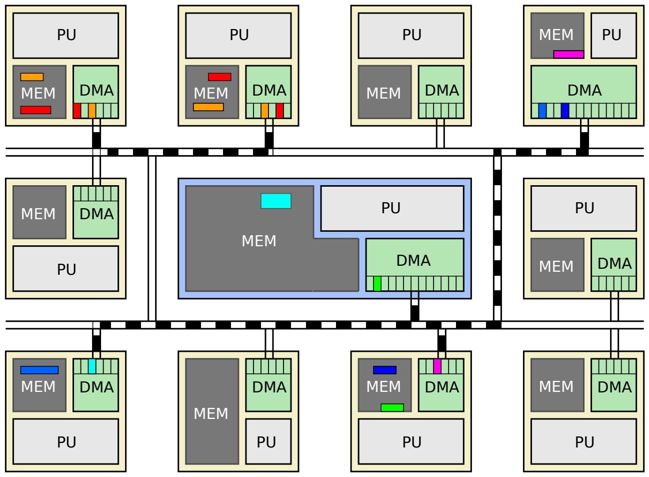

Figure 1: Many small and heterogeneous cores with processing units (PU), local scratchpad memory and DMA units (green) for data transfers over a network-on-chip. The blue core in the middle is responsible for controlling which DMA unit is allowed to send data to which other cores.

The rest of the paper is structured as follows: Section 2 explains how the envisaged and simulated hardware looks like, followed by Section 3 which describes the design of the simulator and especially the DMA interface. Section 4 uses this as a foundation to explain the IPC mechanism that we have implemented. Finally, we describe the performed evaluation, followed by related work and the conclusion.

2. ENVISAGED HW ARCHITECTURE

Figure 1 depicts the hardware architecture that we envisage for future manycore systems. It is based on the Tomahawk II signal processor [6]. The compute cores (yellow) consist of a processing unit (PU), a private scratchpad memory and a *DMA unit*, which is connected to an on-chip network. The DMA units contain *channels* that describe an area in a remote core (indicated by the colored rectangles in the DMA units and the memory blocks in Figure 1) to which they can write. A controlling core (blue) in the center of the picture coordinates the interaction between the cores by setting up appropriate communication possibilities.

DMA units are small autonomous processors to accelerate copying and scatter gather functionality. They are typically used for bulk data transfers between the CPU and devices such as network cards and disks. In architectures such as the one described above, their role is extended to realize the interaction between otherwise separated cores. In other words, they push messages into the memory of cores that run collaborating applications or servers and they pull data from other cores and from the main memory. Since there is no shared memory between the cores, the DMA units provide the only mean to communicate with other cores. Thus, isolation in terms of separate address spaces realized through energy demanding and difficult to predict memory management units is thereby replaced by controlling the access of DMA units to the memory of remote cores.

3. DESIGN OF THE SIMULATOR

This section explains first how we simulate the just described hardware and discusses afterwards the limitations of this approach.

3.1 General structure

The simulator is constructed by running Linux on a cache coherent x86 manycore system and using Linux mechanisms to mimic the described hardware. Since we are only simulating the principle behavior of the hardware, we can achieve nearly native performance when taking unmodified Linux applications and running them on our simulator. This gives us the opportunity to analyze and evaluate the design of the hardware and the operating system running on it. The performance of our simulator allows us to use long lasting benchmarks.

For our prototype, we use one Linux process to simulate one core of the hardware. Each process consists of two threads: one thread pinned to an “odd” core corresponds to the application core of our simulated system and the other one pinned to a nearby “even” core realizes our hardware extension — a DMA unit for core-to-core transfers. That is, the second thread simulates the DMA unit in software. In the optimal case, the host should have at least $2 * n$ cores, if n is the number of processes. But this is not required. In order to use the facilities of the simulator, the applications are linked against a library that provides the DMA unit thread and routines for communication.

The simulator uses an application called *manager* (running on the controlling core), that is responsible for bootstrapping the application cores, loading the desired applications on them, handling simulator calls and hosting simulated devices. At the moment, only a timer and a rudimentary keyboard is implemented. Both run in a dedicated thread and wait until either the timer should fire or a key has been pressed. In both cases an interrupt is delivered to a previously registered application in form of a UNIX signal. That is, a device driver can be implemented by registering to the manager for the reception of interrupts, announcing the signal handler and waiting for signals.

As a shortcut to running unmodified Linux applications directly on our simulator and to evaluate their use on the new hardware for certain operating system functionality (such as file system calls), we implemented a shared library that is loaded via `LD_PRELOAD`. It is responsible for setting up the connection to the manager and intercepting selected system-calls. If the application is specifically written to run on the simulator, it can initiate it's environment explicitly. This means, it can call a library routine at the beginning of the `main` function and `LD_PRELOAD` is not needed.

3.2 Simulating DMA

Our goal is to keep hardware as simple as possible and add only functionality that is inevitable for the performance of our envisaged architecture. In this case, this serves the additional goal of keeping the simulator generic and not tied to a specific and sophisticated piece of hardware. As is illustrated in Figure 2, the DMA unit consists of a configurable amount of so called *channels*, whereas each channel describes a piece of memory of a different core. Furthermore, it has registers

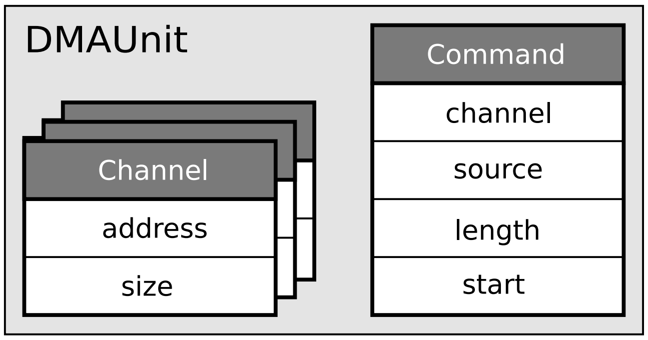

Figure 2: The simulated DMA unit.

to specify the destination slot, the source memory area and to start a transfer operation. For isolation purposes, access to the channel registers can be limited to specific cores (e.g., to the controlling core).

To simulate the behavior of DMA units, we use a dedicated thread, which polls the command registers. These are located in the local memory of the simulated core, while the channel registers are placed in a memory region that is shared with the controlling core, but writable only for the latter. Of course, DMA units local to a core cannot implement push semantics to remote cores because high-end architectures typically lack a means to load data into remote caches. Instead we establish shared memory between the processes and thus between the simulated cores, and the DMA thread copies the data from the memory of the sending core to the shared memory area. That is, we rely on the cache coherency protocol to fetch the data into the remote cache. In this way, our simulated scratchpad memory works more along the line of ARM's overlay mode, where scratchpad and main memory reside at the same address and where a DMA unit transparently fetches addressed locations into the former.

3.3 Discussion

As described, we only simulate the principle behavior of our envisaged future hardware. This makes the simulator suitable for all architectures where the cores do not have a shared memory, but an hardware mechanism like the depicted DMA unit to communicate from core to core.

On the other hand, since we do not simulate the hardware details like the ISA and the timing behavior, but run the applications natively on x86, we can not achieve the accuracy that instruction-level or cycle accurate simulators reach. As mentioned in the introduction, we chose this approach as an initial step towards our goal because it allows us both quick and often changes and also nearly native performance for application-level benchmarks. The former is also supported by the fact that our simulator is small and easy to build (less than 2K SLOC).

The application-level benchmarks enable us for example to answer questions like "What is the optimal number of channels?". On the one hand, the number should not be too small because otherwise multiplexing them between different connections to other cores becomes too expensive. On the other hand, in order to stay in line with our goal of keeping the hardware minimalist, we strive to have not more than necessary.

Additionally, the results of application-level benchmarks allow us an early estimation of the performance on the envisaged hardware. This can be done by putting the time for communication and typical compute intensive tasks on both the simulator and the envisaged hardware in relation. Thus, this approach does not require porting a complex software stack to the new hardware.

4. IPC OVER SIMULATED DMA

To evaluate our simulated DMA unit and to demonstrate the co-design of hardware and operating system functionality, we have implemented an IPC mechanism that facilitates our DMA units for data transfer.

4.1 General IPC Design

Because at the lowest level, DMA is an asynchronous transport mechanism, we decided to realize IPC in an asynchronous and non-blocking manner as well. That is, upon sending a message, there is no entity blocking the sending thread until the message is received by the recipient. Instead, if synchronous behavior is required at the application-level, it has to be implemented on top of the asynchronous DMA transfer. To support that, the sender needs to have a mechanism that allows him to check or get notified when the receiver has processed the request. Therefore, we have decided that each IPC operation consists always of a request *and* a response. This allows the sender to wait for the response which implies that the receiver has got and processed the request.

For these reasons we chose to make the IPC mechanism rely on a concept we call *gate* which describes a connection between two applications. Each gate consists of two channels, i.e. one for each participant, and can only be used in one direction, which means that only one party can send requests and the other party has to wait for requests. If a bidirectional communication is required, two gates should be used.

Since the DMA unit is kept simple, most parts of the IPC are left under software control. That is, the software checks for received data and supports different message sizes over a simple block-wise transfer medium. To achieve that, each IPC message consists of a body and a trailer, whereas the trailer contains the length of the body and the state (read or unread). To send a message, the length of the body is written to the trailer and the state is set to unread. This way, the receiver notices that he has got a new message as soon as the message has been transferred. Since copying by the DMA unit is expected to be done from the beginning to the end, the trailer, as the name suggests, is received after the data-part. Furthermore, the message is put at the end of the shared memory region. Those two properties lead to both a fixed location to check for new messages and ensure that a message has always been completely transferred as soon as the receiver has noticed that the state is unread.

As already mentioned, the DMA unit has a fixed amount of channels. To reach our goal of keeping the required chip area as small as possible, we strive to use only the minimum number of channels. This means, the software might need more gates to other applications than channels are available. Of course, the application programmer should not need to deal with this problem. To make it transparent for him, the gate abstraction ensures that the gate that is to be used gets mapped to a DMA channel. If the application wants to send something over a gate, the abstraction checks at the beginning whether the gate is mapped to a channel. Otherwise, a call to the manager is performed to let the manager configure the specified DMA channel for the gate. This means, the application assigns channels to gates on its own, but can not arm them. It might also be necessary to choose a victim channel if all channels are currently in use.

A problem that remains is that the sender could theoretically copy data to the receiver all the time. Since the receiver should not be required to trust the sender, he has to be prepared for that. One way is to first copy the request to a different location and only use this one afterwards. A different way would be to have a lightweight hardware-mechanism to disable transfers during that time (to save memory and time). As this alternative would require more complicated flow-control logic, we evaluated the first approach and compared it to the best case of avoiding all unnecessary copies when we invoke the operating system functionality locally.

4.2 IPC on the Simulator

On the simulator, each channel is backed by a shared memory area which is mapped into the virtual address space of both participants. That is, a gate consists of two such areas, one for each direction.

To make more clear in what steps an IPC operation is performed and how these map down to the simulator, the following gives an example, illustrated in Figure 3. In step 1, the CPU thread of simulated core 0 tells its DMA unit that it should send a request to the green channel. Afterwards, in step 2, the DMA unit performs the corresponding `memcpy` to the green shared memory area. As soon as the transfer is completed, the CPU thread of core 1 notices during polling that the state has changed and processes the message (step 3). In step 4, it notifies its DMA unit to send the reply back which is done in step 5 via a `memcpy` to the red shared memory area. Finally, in step 6 the CPU thread of simulated core 0, which has polled since sending the request, notices a state change and has thus received the reply.

5. EVALUATION

To evaluate our simulator, we redirected selected system-calls to a different Linux application via our IPC mechanism. This application performs the system-call and transfers the result back to the calling application, again via IPC. In order to prevent changes to existing Linux applications, this redirection is achieved by implementing a shared library that is loaded by an application via `LD_PRELOAD`. This library causes system-calls to not directly call the kernel, but jump to a piece of code of the library. Linux uses Thread Local Storage (TLS) to store the location of the system-call entry code. Thus, we can simply set the corresponding TLS value to the address of a desired piece of code to intercept all system-calls. Most of them are simply passed to the original system-call entry. But the ones we are interested in are handled by the library.

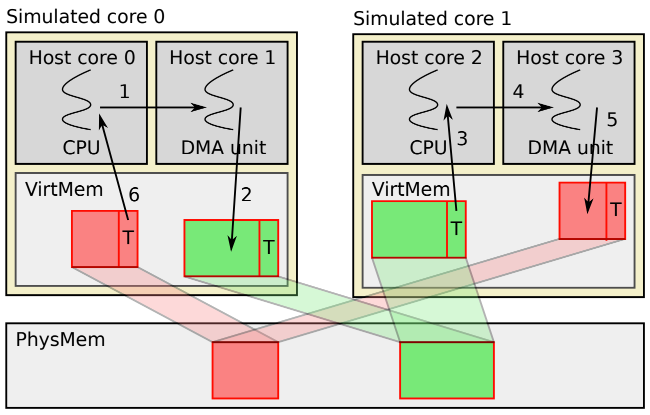

Figure 3: IPC operation simulated on Linux. Each process simulates one core and shared memory regions are used to transfer data between processes. The DMA unit thread is responsible for copying messages, while the CPU thread polls on the trailer (T) until a message arrived.

We chose to redirect the system-calls `read` and `write` for our evaluation, because they are performance critical and often used by applications. To allow an application to perform system-calls for a different application, some other system-calls had to be intercepted and handled as well. These are `open`, `dup`, `dup2`, `fcntl` and `close`. Their parameters are also passed via IPC to the application that should perform `read` and `write` and the library keeps track of the file descriptor mapping (local file descriptor to remote file descriptor).

The first two benchmarks are hand-crafted micro-benchmarks. The first reads files of different sizes from a ramdisk into the application. The second writes files of different sizes from the application to a ramdisk. Figures 4 and 5 show the result of these two benchmarks. The performance when redirecting the `write` or `read` system-call is slower by a factor of 2 to 3 when compared to native execution. The slowdown comes from the fact that we need to perform two additional `memcpy` operations on the data. For example, for reading, the application that performs the system-call first reads the data into a buffer, afterwards the simulated DMA unit copies it to the shared memory area and finally the other application copies it from that location into the buffer that the user prepared for `read`.

For the third benchmark we chose an application-level benchmark. We took the source of the Linux 3.7.6 kernel (545MB) and copied it via `cp -R <src> <dst>` to a different folder. Again, this has been done using a ramdisk. The native copying took 1.15 seconds while the simulated one, that redirected both `read` and `write`, needed 3.45 seconds. As with the other two benchmarks, this is the average of 20 runs with a standard deviation below 3%.

All benchmarks have been done on an Intel dual socket machine with two Xeon CPUs running six physical cores each at 2.6 GHz. We used Linux 2.6.32 as host for our simulator.

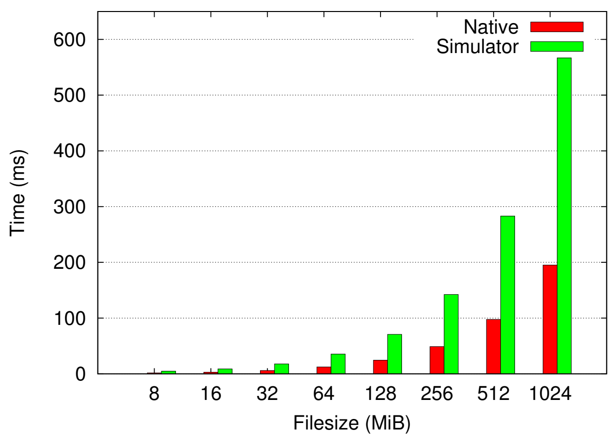

Figure 4: Reading files with varying sizes. It shows the average over 20 runs with a standard deviation below 2.5%.

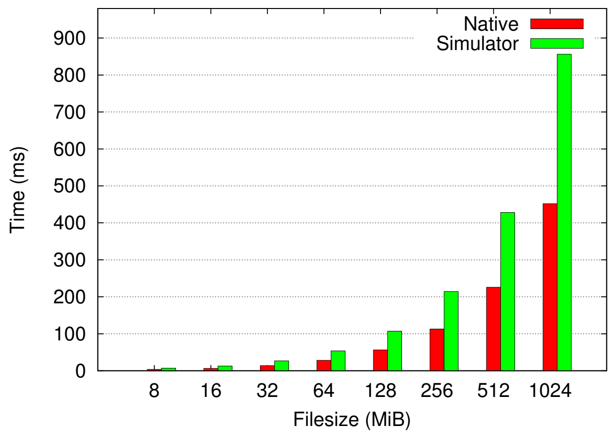

Figure 5: Writing files with varying sizes. It shows the average over 20 runs with a standard deviation below 2%.

6. RELATED WORK

Hardware-software co-simulation can be achieved in many different ways, whereas each of them has a different trade off between performance, accuracy and debugging opportunities [11]. Cycle-accurate simulators like the Xtensa Instruction Set Simulator ISS [13] simulate every cycle of the hardware and achieve thus a very good accuracy, but with a slowdown of three orders of magnitude. This type of simulator eases the analysis of the hardware, but is typically too fine grained to debug the software. Instruction level simulators as for example the Xtensa TurboXim [13], Simics [5] and QEMU [3], experience less slowdown (one to two orders of magnitude), often realized by using binary rewriting techniques, but have also less accuracy than cycle-accurate simulators. They allow a mediocre analysis of both hardware and software. We have chosen a third type, called "virtual hardware" [11], whereas no hardware model is used and software is directly executed on the host. Thus, the accuracy is very low, but the slowdown as well. Additionally, it does not allow to debug the hardware, but supports an easy analysis of the software due to the large amount of available debugging tools on the host.

Similar to our work in terms of focusing on multicore simulation is the distributed parallel simulator Graphite [10]. It uses a binary rewriting technique to execute the software directly on the host, but inserts calls to the simulator at certain points. In contrast to our approach, the ISA of the target architecture is simulated and the slowdown compared to native execution is larger (three orders of magnitude).

Another related approach is described in [7]. Similarly to our simulator, it uses UNIX abstractions to simulate the hardware. It does not execute the software natively, but inserts calls to a function named `delay` to simulate the timing of instructions of the target architecture.

Of course we are not alone in our broader mission of finding new ways for constructing systems by using many simple and heterogeneous cores that interact with each other via light weight hardware mechanisms. For example, similar directions are followed by new kernel designs such as NIX[8], Fos[14] or Barrelfish[9]. They suggest a partitioned design where each core is dedicated to a specific application and communication between cores is achieved by explicit message transfers. Such designs can easily support heterogeneous cores and do not require large and complex cores but benefit rather from a large number of simple cores. Because of the small and heterogeneous cores, the performance and energy saving potentials are huge.

Last but not least, there are several existing IPC mechanisms. For example, Barrelfish [9] consists of multiple kernels running on different CPUs that communicate via message passing over shared memory. Another example is the IPC mechanism in the L4 microkernel Fiasco.OC [2], which also relies on shared memory. A different example is the microhypervisor NOVA [12] which does not allow cross core IPC but only a rudimentary form of signaling. In contrast to them, our approach is to build upon a dedicated hardware mechanism that allows explicit data transfers from core to core. In particular, we do not require system calls to invoke our IPC mechanism.

7. CONCLUSIONS

In this paper, we have seen how high-end manycore platforms can be turned into simulators of future manycore systems with near native application performance. We have shown our simulation of an architecture without shared memory, but local scratchpad memories and DMA units for the communication between cores. Using "odd" cores as application cores and "even" cores to simulate the DMA units, we demonstrated the usefulness of our simulator for exploring DMA-based system-call forwarding over our inter-process communication mechanism.

In the future, we plan to extend our simulation environment to evaluate several details of replacing shared memory via dedicated memories with remote controlled DMA as a secure mechanism for overcoming the implied isolation. In particular, we are confident that questions like the number of channels per DMA unit can be answered when simulating realistic workloads at near realistic speed.

8. ACKNOWLEDGMENTS

This work is in part funded by the European Union and the state Saxony through the ESF young researcher group IMData.

9. REFERENCES

[1] The cell architecture. http://domino.research.ibm.com/comm/research.nsf/pages/r.arch.innovation.html. last checked: 02/08/2013.

[2] Fiasco.oc. http://os.inf.tu-dresden.de/fiasco/overview.html. last checked: 02/08/2013.

[3] Qemu - open source processor emulator. http://wiki.qemu.org/. last checked: 02/08/2013.

[4] Simplescalar llc. http://www.simplescalar.com/. last checked: 02/08/2013.

[5] Wind river simics - full system simulation. http://www.windriver.com/products/simics/. last checked: 02/08/2013.

[6] O. Arnold, B. Nöthen, and G. Fettweis. Instruction set architecture extensions for a dynamic task scheduling unit. In *IEEE Annual Symposium on VLSI (ISVLSI'12)*, August 2012.

[7] M. Bacivarov, S. Yoo, and A. A. Jerraya. Timed hw-sw cosimulation using native execution of os and application sw. In *High-Level Design Validation and Test Workshop, 2002. Seventh IEEE International*, pages 51–56. IEEE, 2002.

[8] F. Ballesteros, N. Evans, C. Forsyth, G. Guardiola, J. McKie, R. Minnich, and E. Soriano-Salvador. Nix: a case for a manycore system for cloud computing. *Bell Labs Technical Journal*, 17(2):41–54, 2012.

[9] A. Baumann, P. Barham, P. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44, 2009.

[10] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12. IEEE, 2010.

[11] J. A. Rowson. Hardware/software co-simulation. In *Design Automation, 1994. 31st Conference on*, pages 439–440. IEEE, 1994.

[12] U. Steinberg and B. Kauer. Nova: a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems*, pages 209–222. ACM, 2010.

[13] Tensilica Inc. Xtensa instruction set simulator (iss) and turbo xim fast functional simulator. http://www.tensilica.co.jp/products/sw_iss.htm, Feb. 2013.

[14] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, Apr. 2009.