Simulating Server Consolidation

Antonio García-Guirado, Ricardo Fernández-Pascual, José M. García[1]

Abstract— **Recently, virtualization has become a hot topic in computer architecture research. The cost reduction and management simplification brought by server consolidation are good reasons why virtualization has become so popular. But there is a lack of tools for researchers to seek new proposals of architectures that improve the performance of virtualized systems. To fill this niche we have developed Virtual-GEMS, a multiprocessor simulator that allows us to simulate the behavior of a virtualized system and research new architectures suitable for virtualization. For testing Virtual-GEMS, we describe and evaluate some configurations for the shared L2 cache of a 16-core CMP running 4 virtual machines.**

Our main contribution is the ease of configuration of simulations of virtualized workloads. Virtual-GEMS uses ordinary system checkpoints as virtual machines. This way, it avoids the need to create complex checkpoints including the hypervisor and the images of the virtual machines to simulate.

I. Introduction

Nowadays, full system virtualization is receiving renewed interest after years of relatively little activity. One of the main reasons is its application to server consolidation, which is the most important use of virtualization today.

If only one application is being run in a big server, then it is probably underutilized because most applications usually do not have enough parallelism to use all the processors or do not require all the available resources. Additionally, without server consolidation, a typical data center is made up of several small servers for executing different services, making the management more complex and expensive.

The solution to this is server consolidation, achieved through virtualization. In short, consolidation brings several servers into a single big server. This cuts management costs, as the number of machines to manage and maintain gets reduced. In contrast to executing every application on the same operating system instance, using a VM (*Virtual Machine*) for each service provides more flexibility (for example, the ability to use different operating systems for different applications) and more security, isolating each application from the others.

Simulation is used in computer architecture research to validate and evaluate new proposals. For example, simulation can be used to determine which is the best branch prediction scheme or to test new coherence protocols. It would be logical to use simulators for also determining which architectures or architecture parameters are best for a virtualized system. However, the simulators available do not provide adequate mechanisms for researching in this field in an easy way.

[1]Dpto. de Ingeniería y Tecnología de Computadores, Univ. Murcia, e-mail: {toni, rfernandez, jmgarcia}@ditec.um.es

In this paper we present Virtual-GEMS, a simulation infrastructure that actually provides the ability to simulate virtual machines. Virtual-GEMS is based on Simics [1] and GEMS [2] simulators, and it provides full-system virtualization, i.e., each virtual machine runs its own operating system instance.

The rest of the paper is organized as follows. In Section II we give some background. Section III explains the development Virtual-GEMS and its structure. In section IV we describe the tests performed to show the use of the simulator and its benefits. Finally, in sections V and VI we present some proposals of future work and our conclusions, respectively.

II. Background

A. Related work

Research in the implications of server consolidation for computer architecture has grown over the last years. Enright et al. [3] showed that server consolidation raise interactions across the consolidated workloads, which open new paths to research. They demonstrated that workloads cannot be evaluated just in isolation when researching consolidated environments.

Marty et al. [4] propose two new two-level coherence protocols. These protocols actually provide a virtual cache hierarchy adaptable to the virtual machines executing in the consolidated server, achieving considerable performance improvements in relation to traditional protocols. One important topic in that paper is that their protocols are well suited for inter-VM page sharing. However, their tests do not simulate this feature.

In the work of Apparao et al. [5], an analytical performance model for consolidated workloads is developed, instead of using simulation.

Finally, Hsu et al. [6] explore the cache design space for CMPs by using traces instead of full system simulation.

B. Basic simulator infrastructure

We base our simulator on GEMS [2], a multiprocessor simulator developed by the Multifacet Project from the University of Wisconsin-Madison. It is based upon Simics, a full system simulator released in 2002 and developed by Virtutech.

Simics [1] is a simulator accurate enough to execute unmodified operating systems and even device drivers. It focuses on functional simulation, and can be expanded in many ways by building modules that give it new features.

GEMS extends Simics so that it can, among other things, simulate the timing of a detailed and configurable memory system. For that purpose, GEMS implements several Simics modules. One of them,

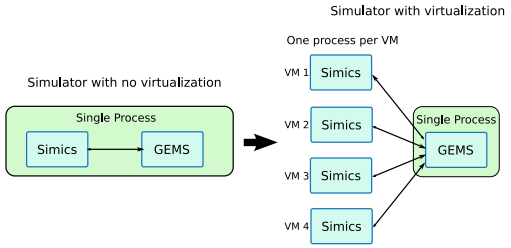

Fig. 1. *Evolution from simulating a single real machine to simulating a virtualized system.*

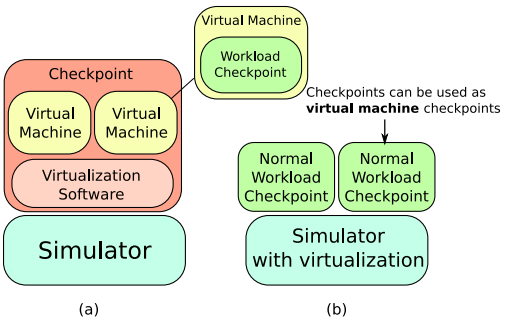

Fig. 2. Virtualization Schemes. *With in-simulation virtualization software very complex checkpoints must be used (a), while with virtualization inside the simulator the already available checkpoints of the workloads can be used as VMs (b).*

Ruby, is the responsible for simulating the memory system.

This modular simulation infrastructure decouples functional simulation (driven in Simics) and timing simulation (driven in GEMS).

III. Virtual-GEMS

Virtual-GEMS intends to simulate several virtual machines. It is based on Simics and GEMS simulators as its basic constructing blocks. Virtual-GEMS simulates each virtual machine by means of a Simics instance (only functional simulation), each one with its own OS and workload. But in order to get a single view of the whole server, a single GEMS instance is used to simulate the timing of all virtual machines. To do that, we first decouple Simics and GEMS, and then we develop a mechanism to connect several Simics instances (i.e. virtual machines) to the same GEMS timing simulation (see Figure 1). This way, we approximate the behavior of a virtualized system. We do not simulate any software virtualization layer, instead, in the current implementation, we assume that every virtualization issue is managed in hardware. The design of Virtual-GEMS allows us to implement the functionality of the hypervisor as part of the simulator.

The first decision is to choose the more adequate virtualization scheme. Virtualization software could be executed directly inside the simulation. If in-simulation virtualization software were used, to create a checkpoint we would have to include the virtual machine images inside the new checkpoint. And these virtual machine images are in turn checkpoints of the workloads to be simulated. Modifying any workload or any virtualization software parameter would require to build a new checkpoint, and creating such a complex checkpoint is a time demanding task.

The other approach is to implement virtualization in the simulator itself, allowing the direct use of the workload checkpoints already used for simulations that do not involve virtualization. This way, changing the workloads to use in each virtual machine or changing the virtualization parameters (which would be set in the simulator) would not require the creation of new checkpoints. The differences between both approaches can be seen in Figure 2. We have followed the later approach in our case.

This is the main contribution of our simulation infrastructure. We avoid the need to create a large number of complex checkpoints to perform simulations. We also provide a simple hypervisor inside our infrastructure. In order to include new virtualization features, this in-simulator hypervisor is easier to expand than in-simulation virtualization software. The latter would also require to create new complex checkpoints everytime the virtualization software is modified, whereas our hypervisor can be freely modified and the original checkpoints can still be used.

A. Modifying the Simics-GEMS communication

The Simics simulator provides interfaces to be extended. Users can use these interfaces to extend some missing functionality in the simulator. For example, the memory access timing can be developed in a timing-model interface.

Processors modelled by Simics are simple in-order non-pipelined processors which block at every memory access. GEMS provides a more detailed out-of-order processor simulator in a module called Opal. Due to the core simplification that seems to be the trend nowadays in many-core CMP designs (mainly due to power consumption and heat constraints), we use the Simics simple model in our infrastructure instead of the out-of-order model simulated by Opal.

The most important element in GEMS is the Ruby module, which is responsible for simulating the memory system.

The interface between Simics and Ruby is another module that we call SimicsRuby, which passes and preprocesses the memory requests from Simics to Ruby and returns to Simics the information about completed requests.

The module implementing the timing-module interface (SimicsRuby) receives all the information of each memory request as these are performed by the processors simulated by Simics. This module returns the number of cycles that the requesting processor will stall until the access is completed. The information concerning the memory access that is passed through the timing-module interface includes the memory address, the kind of access (read, write), and other necessary information.

The synchronization between Simics and Ruby is an important aspect. Memory requests do not carry

any information about the cycle in which they are performed. Thus, some additional mechanism is needed to carry out this synchronization.

To do that, a Simics API function is used for registering a callback in Simics. This function is called each time a fixed number of cycles passes. When the callback is executed, the events in Ruby corresponding to that cycle are executed too.

Unfortunately, when a memory request is issued, we cannot predict how many cycles it will take to complete. Instead, the actual implementation blocks the requesting processor for a huge number of cycles, such as to be sure that the request will be completed before that processor wakes up. As the simulation advances, the events in Ruby keep executing each cycle, and eventually the memory access that blocked the processor will be completed. Then, Ruby will unblock the requesting processor (through SimicsRuby) so it can continue its execution. This way, the requesting Simics processor stays blocked from the moment it sends the request until the time the request is satisfied. This mechanism feeds back the timing information to the functional simulator.

As an intermediate step in the construction of Virtual-GEMS we have divided the SimicsRuby module to decouple Simics and Ruby. Two new modules have been developed. The first one is the module called mi-device, which implements the timing-model interface and receives the Simics memory requests. All the Simics dependent code has been put into this module (e.g. all the calls needed to perform the synchronization).

The second module is referred to as RemoteRuby, and contains all the calls to Ruby elements. It communicates through a pipe with mi-device using a new and very simple interface which is used to send all the information concerning memory accesses and synchronization.

This way, we move from a single process executing Simics and GEMS to two different processes: one for Simics and other for GEMS, communicating through FIFO pipes.

B. Virtualization in Virtual-GEMS

First of all, we need to remind that our focus is on full-system virtualization. Each virtual machine runs its own operating system instance.

The virtualization process has to handle three main elements: processors, memory, and I/O devices. In our scheme, processors are physically partitioned so each virtual machine runs in a subset of the processors of the real machine. I/O buses are also physically partitioned, so each VM has its own set of disks and other devices. Memory is logically partitioned. The information of different VMs is interleaved in memory with a memory page granularity. This scheme is depicted in Figure 3.

Virtual memory is a key point to be properly handled in the virtualization context. Broadly speaking, virtual memory is supported by a page table for each process that maps the *virtual memory addresses* handled by each process to *real memory addresses* which can be found in memory or in swap space in the disk. This page table is usually handled by the OS.

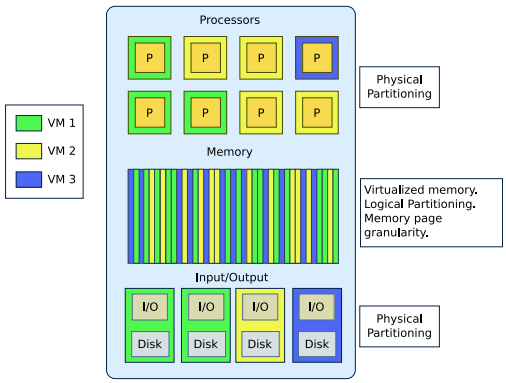

Fig. 3. Virtualization Scheme. *Physically partitioned processors. Page level memory virtualization. Physically partitioned I/O.*

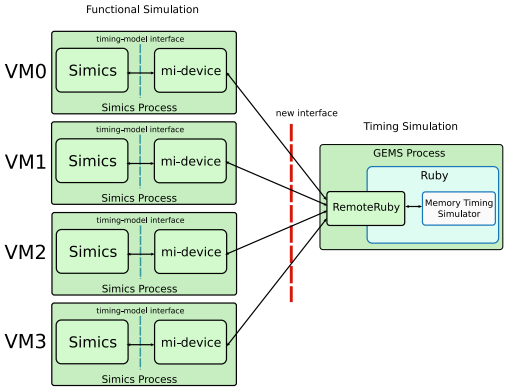

Fig. 4. Structure of Virtual-GEMS. *Each VM is supported by a Simics instance. The timing simulation of the whole system is performed in GEMS. RemoteRuby manages all the virtualization issues.*

In the virtualization environment, an extra level of page tables is needed. This extra level of page tables is managed by the hypervisor, and contains a page table for each VM. This page table maps the *real memory addresses* accessed by a VM to *physical memory addresses* in the physical machine. This is an approach similar to the one taken in Cellular Disco [7].

To build this virtualization scheme, Virtual-GEMS uses different Simics instances as VMs (see Figure 1). Each Simics instance is a different process, simulating a complete virtual machine in a functional manner, with its own processors and I/O devices. On the other hand, a specific subset of the processors simulated by Ruby corresponds to each VM. Therefore, we need to do a mapping between Simics processors and Ruby processors. We also need to map real memory pages from each VM to physical pages in the real machine using the new page table level mentioned before. All these virtualization issues (which would be handled by the hypervisor in a real system) are implemented in the RemoteRuby module which also controls all the concurrency and synchronization issues between VMs. Figure 4 shows the final structure of Virtual-GEMS.

C. Synchronization and parallelism

We force the execution of all the Simics instances to advance steadily. To do that, every mi-device module attached to a Simics process sends a trigger-event message to RemoteRuby every fixed number of Simics cycles, and then stops and waits for confirmation from RemoteRuby. Once RemoteRuby has received the trigger-event messages from every mi-device module, the virtual machines are synchronized. Then, RemoteRuby executes the events corresponding to that cycle and then sends the confirmation messages to the mi-device modules. The Simics instances can now continue the functional simulation of the virtual machines. With this process, the execution of every virtual machine advances one Ruby cycle. The equivalence between Simics cycles and Ruby cycles is customizable and it is used to poorly approximate an N-way superscalar processor by using the Simics simple processors, where N is the value of the cycle multiplier. Therefore, all the memory requests performed by the Simics instances between two RemoteRuby confirmation messages are considered to be issued in the same Ruby cycle. This process is the same as the one performed in the original GEMS, but it now involves several Simics instances instead of a single one. Thanks to the trigger-event and confirmation messages, the synchronization and advance of all virtual machines does not introduce any loss of accuracy into the simulation.

In contrast to the original GEMS, where the whole simulation was performed in a single thread, Virtual-GEMS shows some level of parallelism. The functional simulation of each VM is performed in a separate Simics process, allowing each VM simulation to execute in a different core of our simulation servers. Unfortunately, the heaviest part of the simulation corresponds with the timing simulation performed by Ruby, which is single threaded.

Therefore, Virtual-GEMS can speed up the simulation when executed in a multicore server, i.e., allowing several parallel simulations of few-core VMs instead of one single simulation of a many-core full system. On the other hand, the communication between processes introduces an important overhead, reducing the benefits of the parallelization.

IV. Testing the simulator

Once the structure of the simulator has been shown, in this section Virtual-GEMS is used to evaluate the best L2 configuration for a multicore architecture in which virtualization is used. The main purpose of this evaluation is to test the simulator and check the sanity of the results.

A. Simulated Architectures

Virtual-GEMS allows the simulation of a wide range of machine configurations. Our base architecture is a tiled-CMP with 16 tiles. Each tile has a processor, a private L1 cache, and a bank of the shared L2 cache. Since the L2 is shared, cache coherence must also be kept at L1 level.

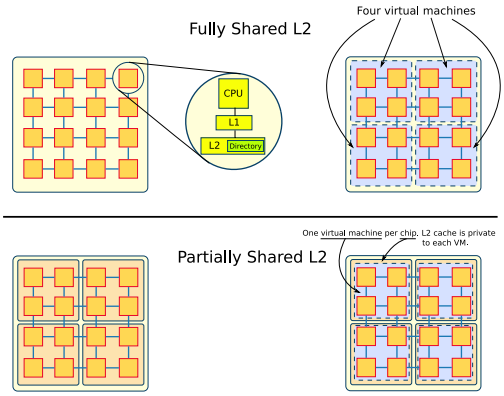

Fig. 5. Simulated Architectures. *The tiled CMP with Fully Shared L2 (above) and the Virtual Machines placement on it. The Partially Shared L2 configuration (below) and VMs placement.*

On top of this base architecture, we set up four virtual machines, with four tiles each. We evaluate two different configurations that we derive from our base architecture, and we call them *Fully Shared L2* and *Partially Shared L2*. In the *Fully Shared L2*, all the L2 banks are shared among all the cores in the chip, regardless of which virtual machine they belong to. In the *Partially Shared L2*, the L2 banks are only shared by the tiles of a specific virtual machine. Therefore, L2 banks are not shared among VMs but they are private to each VM. The simulated architectures are shown in Figure 5.

The placement of data lines in L2 cache banks is determined by a subset of the bits of the physical address. In the case of *Totally Shared L2* these bits choose the specific bank among all the banks of the chip. On the other hand, for *Partially Shared L2*, the bits choose the specific bank among those private to the VM. In both cases, these bits are out of the page offset part of the address, hence the hypervisor has control over them when it performs the mapping from real address on the VM to physical address.

We take advantage of this fact to try to place data as close as possible to the cores that will use them. To do this, the hypervisor chooses the physical address that will correspond to each real address of each VM so that the memory lines of that page will map to the desired L2 bank.

We have used three different real-to-physical address mappings based on the proposals of Cho et al. [8]. The first mapping is a simple arrival order assignment, which maps a new real memory page from a VM to the next free physical page. This approach, called *simple mapping*, does not control where the data is located in the chip (i.e. in which L2 cache bank), so we consider it as an almost random assignment. The second one, called VMCBM (*Virtual Machine Cache Bank Mapping*), maps all the real pages from one VM to the L2 banks belonging to that VM, using a round-robin order to choose the L2 bank inside the VM. The third mapping is referred to as TCBM (*Tile Cache Bank Mapping*) or first-touch policy. It brings the data to the L2 bank of the tile whose core first accessed the page.

Name	Architecture	Real-to-Physical Address Mapping
config1	Fully Shared L2	simple mapping
config2	Partially Shared L2	simple mapping
config3	Partially Shared L2	VMCBM
config4	Partially Shared L2	TCBM

TABLE I
MACHINE CONFIGURATIONS TESTED.

Processors	16 UltraSPARC-III+ 4-ways, in-order. 2 GHz
L1 Cache	Split I&D. Size: 128KB. Associativity: 4-ways. 64 bytes/block. Access latency: 2 cycles.
L2 Cache	Size: 1MB each *slice*. 16MB total. Associativity: 4-ways. 64 bytes/block. Data array access latency 15 cycles.
TLB	64 entries, totally associative
RAM	4 GB DRAM. 1 memory controller for each chip. Memory latency 160 cycles + on-chip latency
Interconnection	Bidimensional mesh 4x4. 64 bytes links. 8 latency cycles by link.

TABLE II
COMMON TARGET ARCHITECTURE CHARACTERISTICS.

Workload	Description	Size	Simulation
apache4x4p	Web server with static contents	3000 transactions per VM	Four 4-processor Apache VMs
jbb4x4p	Java server	5000 transactions per VM	Four 4-processor JBB VMs
unstructured4x4p	Fluid dynamics application	Mesh.2K, 5 time steps	Four 4-processor Unstructured VMs
barnes4x4p	Simulation of gravitational forces	8192 particles	Four 4-processor Barnes VMs
commercial	Mix of commercial workloads	See Apache & JBB sizes	Two 4-processor Apache VMs, Two 4-processor JBB VMs
scientific	Mix of scientific workloads	See Unstructured & Barnes sizes	Two 4-processor Unstructured VMs, Two 4-processor Barnes VMs

TABLE III
WORKLOAD CONFIGURATIONS TESTED.

We set up four different configurations that can be seen in Table I. From now on, we refer to the four configurations used as config1/2/3/4. The common characteristics of the architecture can be seen in Table II.

Finally, for the configurations tested, we use two different memory coherence protocols, both based on a MOESI state scheme. The first one is a two level directory based protocol [9]. The second protocol is a token-based one [10].

B. Workloads

We use a mix of commercial and scientific workloads for testing Virtual-GEMS. We use two commercial workloads: the Apache web server and the JBB Java server. For the scientific workloads we use one of the SPLASH-2 suite benchmarks [11] called Barnes, and another scientific benchmark called Unstructured. The details of these workloads can be found in Table III.

C. Methodology

In the virtual machine environment, with a chip running four different VMs, we need to define a metric for measuring the performance of the full system. One alternative is to run the simulation for a fixed amount of time, and then count the transactions (i.e. units of work) completed by the workload in each VM to measure the performance. But this approach has several problems, as the execution of scientific workloads (Barnes and Unstructured) cannot be easily split in transactions. Moreover, since a transaction is a unit of work which depends on the particular benchmark, counting transactions for a workload made of a mix of different benchmarks is not straightforward.

Hence, we have decided to fix the amount of work that each workload performs for the different evaluated configurations. We have balanced the size of the different workloads so that every workload take approximately the same time to complete. We use the average number of cycles elapsed by all the VMs in the simulation to complete their workloads as the performance metric regarding the whole system.

D. Results

We use config1 as the base configuration to compare the rest of the configurations that have been proposed in this work. Config1 is the base situation of a CMP architecture with no special adaptation to virtualization or multiprogramming, while the rest of configurations try to improve the performance of this one by using the real-to-physical address mapping policies explained before. The results of these evaluations are shown in Figures 6 and 7.

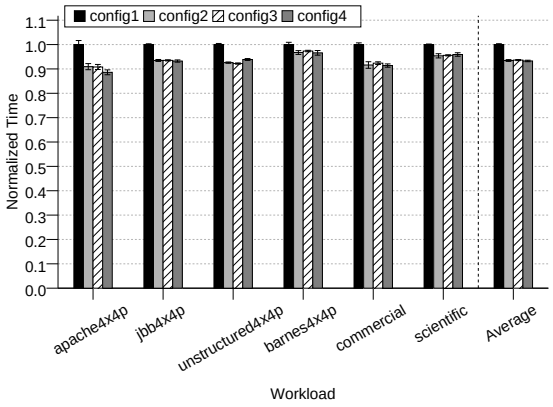

Fig. 6. Test results using the directory-based protocol.

All config2/3/4 get average speedups between 5% and 6% when using the directory based protocol, and around 7% when using the token based one.

The smallest performance improvement in a single benchmark achieved by a *Partially Shared L2* configuration is 2.6% (config3 in Barnes using directory), with the rest of the results showing better

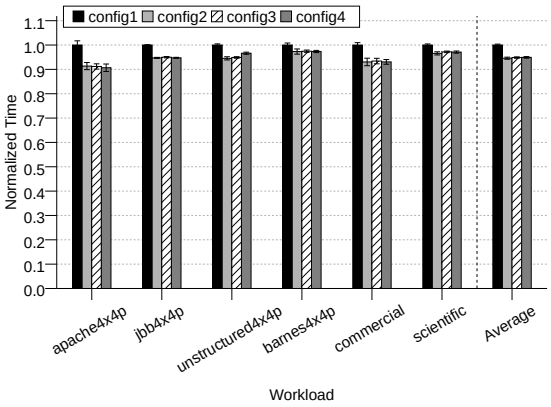

Fig. 7. Test results using the token-based protocol.

improvements. It seems clear that *Partially Shared L2* is a better choice for these workloads than *Fully Shared L2*. Nevertheless, we are not able to point out which one of the three mappings (simple, VM-CBM or TCBM) used in combination with *Partially Shared L2* performs best.

V. Future Work

Implementing page sharing among virtual machines is an interesting path of research. It will provide benefits in the cache usage, specially if the virtual machines run the same OS. On the other hand, the hypervisor introduces overhead since it has to run a mechanism to detect shared pages. It will allow us to reproduce virtual hierarchies [4] actually using inter-VM page sharing to check the behavior of the protocols proposed in that paper.

Another interesting research idea is the development of profiling mechanisms in the hypervisor. With live statistics about the execution we can dynamically reassign the resources of the physical machine to fit the needs of the virtual machines. This can provide support to check the potential for performance improvement of new ideas for dynamic reassignment.

VI. Conclusions

Simulation is an important and extensively used tool in the computer architecture research arena. Virtualization has become a hot topic nowadays, because server consolidation is a good approach to reduce costs and management time. However, there is a lack of simulators usable for research in the virtualization field.

This paper introduces Virtual-GEMS, a new simulator based on Simics and GEMS that allows us to simulate the behaviour of a multicore architecture running several virtual machines on it. Virtual-GEMS is based on virtualizing the functional simulator (by creating several instates of Simics, as many as the number of VMs), and doing the timing simulation with a single instance of GEMS.

The main contribution of our infrastructure is the ease for using ordinary checkpoints as VMs, instead of needing to build complex checkpoints including the hypervisor and virtual machines. Also, the hypervisor included in our infrastructure can be expanded with new functionality without having to create new checkpoints.

We consider that our simulator can help to improve the new multicore architectures designed for being used for server consolidation.

Acknowledgements

This work has been jointly supported by the Fundación Séneca (Agencia Regional de Ciencia y Tecnología, Región de Murcia) under grant 00001/CS/2007, and also by the Spanish MEC and European Commission FEDER funds under grants "Consolider Ingenio-2010 CSD2006-00046" and "TIN2006-15516-C04-03". Antonio García-Guirado is also supported by a research grant from the Fundación Séneca.

References

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

[2] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, November 2005.

[3] Natalie Enright Jerger, Dana Vantrease, and Mikko Lipasti, "An evaluation of server consolidation workloads for multi-core designs," *IEEE Workload Characterization Symposium*, vol. 0, pp. 47–56, 2007.

[4] Michael R. Marty and Mark D. Hill, "Virtual hierarchies to support server consolidation," *SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 46–56, May 2007.

[5] Padma Apparao, Ravi Iyer, and Don Newell, "Towards modeling & analysis of consolidated cmp servers," *SIGARCH Comput. Archit. News*, vol. 36, no. 2, pp. 38–45, 2008.

[6] Lisa Hsu, Ravi Iyer, Srihari Makineni, Steve Reinhardt, and Donald Newell, "Exploring the cache design space for large scale cmps," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 24–33, 2005.

[7] Kingshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum, "Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors," *ACM Transactions on Computer Systems*, vol. 18, no. 3, pp. 229–262, 2000.

[8] Sangyeun Cho and Lei Jin, "Managing Distributed, Shared L2 Caches through OS-Level Page Allocation," in *MICRO*, 2006, pp. 455–468.

[9] Michael R. Marty, *Cache coherence techniques for multicore processors*, PhD in Computer science, University of Wisconsin - Madison, 2008.

[10] Michael R. Marty, Jesse D. Bingham, Mark D. Hill, Alan J. Hu, Milo M. K. Martin, and David A. Wood, "Improving multiple-cmp systems using token coherence," in *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, Washington, DC, USA, 2005, pp. 328–339, IEEE Computer Society.

[11] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proceedings of the 22th International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, 1995, pp. 24–36.