CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2014; **00**:1–16
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Modeling and analysis of performances for concurrent multithread applications on multicore and GPU systems

D. Cerotti[1], M. Gribaudo[1], M. Iacono[2*], P. Piazzolla[1]

[1] *Politecnico di Milano, Dipartimento di Informatica, Elettronica e Bioingegneria, Via Ponzio 34/5, I-20133 Milano, Italy*
[2]*Seconda Università degli Studi di Napoli, Dipartimento di Scienze Politiche, Viale Ellittico 31, I-81100 Caserta, Italy*

SUMMARY

The capabilities of multicore processors lead them to be widely adopted in systems at any scale, since their are able to provide more computing power at a lower consumption and dissipation cost. System designers are challenged to a deeper understanding of multicore functioning in order to fully exploit them while keeping the optimal balance between cores utilization and optimal throughput, response time and energy usage.
Besides the advancement of general purpose CPUs, the same technological evolution leads to the rise of GPUs, dramatic evolution of graphical coprocessors, that are now affordable, efficient, dedicated computing units, capable of parallel computing and equipped with facilities that make them suited for supporting the main CPU of a system in running ordinary applications. The availability of Commercial off-the-shelf (COTS) multicore computer with one or more collaborating GPUs makes them the basic building block of data centers devoted to cloud applications or scientific computing.
The way to optimal exploitation of such a wide amount of computing power passes through the ability of matching the best scheduling of hardware resources with the software characteristics of the applications. This requires appropriate models and evaluation methods.
Simulation and analytical techniques are an essential tool to support the design and the management process of such architectures, but a sound characterization of the workloads is required. Typical workloads consist in multithreaded applications, with different characteristics, that dynamically span over the cores of multiple machines, connected by fast networks.
In this paper we propose several parametric performance models for different configurations of multicore machines, with or without GPU support, running multiple class multithreaded applications, aiming to supply a detailed modeling help for complex data centers.

Received . . .

KEY WORDS: performance modeling; multicore; GPU; multithread applications

1. INTRODUCTION

The evolution of processors led to the current multicore CPUs (Central Processing Units), that are available at low price and are reliably used into data centers with hundreds of Commercial off-the-shelf products (COTS) or dedicated computers. A separated branch of this evolution, that stems from the same hardware technologies but aims to the development of parallel and vector processors dedicated to the execution of complex graphics operations, resulted into the availability of non general purpose but very powerful CPUs, dubbed GPUs (Graphic Processing Units). Modern GPUs are capable, by means of special software libraries, of executing specific parts of the workload

*Correspondence to: Mauro Iacono, Seconda Università degli Studi di Napoli, Dipartimento di Scienze Politiche, viale Ellittico 31, I-81100, Caserta (CE), Italy - E-Mail: mauro.iacono@unina2.it

generated by a software application (e.g. scientific computing), thus offering to general purpose usage their capabilities.

Sophisticated architectures have been designed to coordinate such an enormous amount of computing power, that can only be exploited by means of flexible scheduling and management policies. Such architectures can be optimized in many aspects: power consumption, heating reduction, parallelism and process granularity exploitation, spatial organization and interconnection. The asymptotic goal is the possibility of a smart management of each single core of each CPU and each GPU, in order to properly allocate them to each application that in a certain moment is executing a workload. This is mediated by virtualization, that allows decoupling between hardware resources and software tasks, and provides a layer of flexible allocation at the cost of a small, additional workload.

The complexity of such architectures results in the need for flexible modeling and assessment tools to support their design and management. Proper performance evaluation techniques can provide significant savings in case of very huge systems. In particular, if there is a big number of application threads running on the system, a small increase in performance can be as much significant as many. This is expecially true in massively distributed architectures or Big Data infrastructures.

In this paper a set of models are presented, that support the design of computing systems based on multicore CPUs and GPUs and run multiclass, multithreaded applications. The aim of their analysis, using both analytical and simulative techniques, is to provide a characterization of different HW/SW configurations, in order to build synthetic models, to be used as building blocks in models for more complex architectures.

This paper extends [1], by adding the analysis of GPU effects and by considering the case of multiclass applications, and by introducing a simple modeling formalism that assists the modeler in developing performance models.

The paper is structured as follows: motivations are analyzed in Section 2 and related works are discussed in Section 3. The proposed language is presented in Section 4 and it is then used to consider single workload models in Section 5 and complex load mixes in Section 6. Finally, conclusions are given in Section 7.

2. MOTIVATION

The correct design and management of computer based systems is a crucial factor to achieve the maximum exploitation of the resources needed to keep it in operating condition (energy consumption, cooling, maintenance, administration, and other additional costs). Whenever the goal of the system is to provide computing services to third parties, correct resources allocation and scheduling is fundamental to sustain a variable workload while keeping the maximum efficiency.

Possible mismatches between requested operations and resources management, due to erroneous scheduling decisions, can temporarily prevent the system to fulfill all the requests in the needed terms. This problem can be limited if a valid prediction of the scheduling decision effects is available. Performance models are thus essential to achieve the optimal scheduling of resources among processes.

Complex systems need tools that provide a correct representation of their internal dynamics. This can be obtained by a proper abstract modeling framework, capable of capturing the nature of the system, and a flexible model evaluation technique. There are many different techniques, based on different premises that can be roughly grouped into analytical or simulation based. In the number of traditionally exploited analytical techniques, at least Petri nets and queuing networks should be mentioned, while the most spread approach on the other side is event-based simulation. The choice between the two categories depends on the context, the personal preferences and the skills and habits of the modeler. This choice is generally not a structural limit to the modeling potential, instead, the choice of the most suited technique can help in understanding some essential, hidden aspect of the system. In fact, modeling approaches exist (SIMTHESys [2][3], OsMoSys [4][5], Mobius [6]) that support flexible choices.

DOI: 10.1002/cpe

Whatever the chosen technique, the system growth in complexity causes a parallel growth in the model complexity. When complexity level is low or a modular approach is viable, both the analytical and the simulation based approach can scale up and be used to build fully functional simulations; after a certain limit, the solution is to resort to approximation (or, of course, to completely change the modeling approach, by choosing a more suitable modeling formalism or more efficient analysis tools). Approximation, by the way, may increase the distance between the modeled and the real behavior of the system, thus keeping the needed trust in the model can become difficult.

The true structural limit is a consequence of the fact that, in most cases, the adopted techniques are limited by the need for a detailed characterization, so that it is necessary to adopt a modeling approach that is based on hierarchical models. This can be done, for instance, by parting the system into its layers, or its components, and studying them separately, with proper solicitations that represent the other elements. In this way it is possible to understand their characteristics in detail, analyze the main performance figures and the factors that influence them, and to understand if secondary aspects, if any, can be neglected, with the goal of synthesizing simplified models to be used at a higher level in full respect of the global behaviors.

The authors deal with the problem of complexity in [7], [8], [9] and [10] by means of multiformalism techniques, in [11], [12] and [13] by resorting to mean field analysis based techniques, suitable for systems that are highly modular and regular and have a high number of elements: in both cases, a model hierarchy based approach has been used, with different aims and purposes. In the first group of papers, the hierarchy is used to experiment with component models, to obtain the simplest suitable form that will potentially empower the scale up of composed models, by also exploiting the best modeling formalism for each of them, that can guide in the simplification process; in the second group, in which the problem of scaling up is basically not a limit, the hierarchy is used to identify and derive the parameters of basic building blocks.

This paper belongs to the second group, and aims to provide the results on which the basic building blocks for massively parallel and distributed cloud systems can be designed. In particular, the building blocks are single computing nodes, that are composed by a single thread or multithreaded software layer and a single core or multicore hardware layer, with or without GPUs, that can be managed in different fashions. The analysis is oriented to performances.

Coherently with the approach, the design of the building blocks is a separate problem. The choice of the technique is partially independent from the general framework. In the following, the first choice is to characterize all blocks as state-based systems, that have only exponential transitions and that can be analyzed by both analytical tools and simulations; the expressiveness is then extended by introducing fork and join constructs and non-exponential transitions, to better match a realistic scenario, at the cost of switching to only simulation based techniques due to the excessive growth of the resulting state space.

In order to provide an example of how these models can be used as a standalone tool for system design and evaluation or within a more complex framework, we also propose a simple modeling formalism, that has been developed within the SIMTHESys multiformalism modeling framework.

3. RELATED WORKS

Multicore CPUs systems performance has been analyzed in literature by several points of view. The main part of academic literature usually focuses on the impacts on performance that a single component of the CPUs may introduce. For example, in [14] the authors propose a low-overhead, runtime mechanism that partitions a shared cache between multiple applications depending on the reduction in cache misses that each application is likely to obtain for a given amount of cache resources. L2 cache sharing is also the focus of [15], where the authors investigate the performances of on-chip cache to propose a new architecture for configuring the share of SDRAM among different CPU functions. Moving outside the borders of a single chip, the goal of [16] is to understand how off-chip bandwidth partitioning affects system performance, and how cache and bandwidth partitioning interact.

DOI: 10.1002/cpe

The whole memory hierarchy is the topic of [17], in which the authors analyze and evaluate support for expressing parallelism and locality in programming models for multi-processors with explicitly managed memory hierarchies.

The modeling approach presented in [18], instead, uses data collected from performance counters on two different hardware implementations using hyper-threading processors to demonstrate the effects of thread interaction.

Thread interaction insights may be useful for guiding operating systems scheduling decisions. In particular, different scheduling policies can introduce strong performance impacts, as highlighted in [19]. Here, a new memory scheduling technique is introduced to improve system throughput without requiring significant coordination among memory controllers.

More abstract features, such as virtualization effects, are considered in [20] where a quantitative analysis of virtualization performance are provided to illustrate how server consolidation can benefit from virtualization. In [21] a series of performance models for predicting performance of applications on virtualized systems are presented. Major factors that affect the performance of virtualization platforms, such as the overhead of full virtualization for CPU-intensive and memory-intensive workloads and how different core affinity properties affect the performance of individual virtual machines, are evaluated in [22]. Another performance study, in [23], presents a light weight monitoring system for measuring the CPU usage of different virtual machines including the CPU overhead in the device driver domain caused by I/O processing on behalf of a particular virtual machine.

In recent years, a large body of work has explored how to use GPUs for general purpose computing, sometimes known as G̈PGPU.̈ Highly parallel programmable processor featuring peak arithmetic and memory bandwidth that substantially outpaces its CPU counterpart, the GPU's rapid increase in both programmability and capability has spawned a research community that has successfully mapped a broad range of computationally demanding, complex problems to the GPU. For example in [24], [25] the authors explore the effectiveness of GPUs for a variety of application types. GPU performance is compared to both single-core and multicore CPU performance. However, in [26] it is demonstrated that the phenomenal performance of GPUs compared to CPUs strongly depends on where in the system data resides, and the overhead to move the data to where it will be used, and back again if necessary.

GPUs are designed for computing a large amount of data in parallel. They have a high data transfer bandwidth and a large number of simple cores. However, GPUs lack the ability of saving process execution state, thus they are unable to run two or more programs in a time sharing manner. This lack of quick context switch makes it difficult to virtualize GPU. Many efforts have been made by researcher to solve those issues, exploiting different techniques. For example in [27] a framework for HPC applications that uses hardware acceleration provided by GPUs to address the performance issues associated with system-level virtualization technology is proposed. The goal of [28] is, instead, to design a GPU provision system that combines CUDA programs from different virtual machines and execute them concurrently, so as to support the concept of GPU sharing among virtual machines.

As the general approach is founded onto the definition or the application of benchmarks that are run on real systems to tune analytical or simulative models, in this paper in vivo measures will be used to validate the proposed models, to obtain a reliable base on which more general performance consideration can be carry out (as, e. g., in [29]), and try to get some general indications about the influence of multithreading and multicore on the overall performances of a complex system architecture.

4. TASKS AND SYSTEM DESCRIPTION LANGUAGE

The scenario we would like to consider includes system composed by several multi-core computational nodes, running multi-threaded applications characterized by a set of different stages using different resources. We propose a two level description language that can be used to describe both tasks and systems running them. Figure 1 shows the primitives available in the formalism, and

DOI: 10.1002/cpe

Table I summarizes the properties associated to each element. The first modeling level considers the infrastructure. `Compute` elements represent the computational nodes of the considered system. They can be PC, servers, blades, storage controller, or whatever can be used to service the tasks the system is executing. For this reason, each compute node is characterized by the number of cores (*NCores*) that can concurrently execute the tasks and their speed (*CoreSpeed*). This speed can be either constant, or load-dependent, to account the fact that some interference and locking between the cores might reduce their performance when they execute tasks in parallel. Nodes can also be equipped by a GPU composed of *NGPUCores* cores, each working at a given speed (*GPUCoreSpeed*). `Compute` elements can also perform IO tasks at the speed specified in property *IOSpeed*.

The workload can be either *open*, to model transactional systems, or *closed* to model batch processes. Open workloads are defined by the elements `Arrival` and `Departure`, that respectively specify the points where the tasks enter and leave the system. In particular, the rate at which jobs enter the system is given in property *Rate* of the `Arrival` primitive. Closed workloads are identified by elements of type `RepJob`. In this case, the number of the corresponding tasks is specified in property *N*.

When a task has been executed in a node, it can be routed to another one for further processing, or it can leave the system (for open workloads). The flow of tasks among the nodes is defined by the `Flow` arcs. When more than one `Flow` arc exits a node (either a `Compute`, `Arrival` or `RepJob` element), the *Routing* parameter specifies the selection policy of the next hop: it can be any of the common job routing strategy defined in queueing system, such as random, probabilistic, round robin, join the shortest queue and so on.

The steps required by each tasks are defined in `Workload` elements. This type of primitive must be connected either with an `Arrival` or `RepJob` element with an incoming `Assign` arc, and to a `Compute` node with an outgoing `Assign` arc. The meaning of this syntax is that a task generated either by the input `Arrival` or `RepJob` primitive, will require the workload defined by the `Workload` element from the `Compute` node to which it is connected.

Workloads are described by sub-models specified in an ancillary formalism that defines the steps required by the corresponding tasks. This corresponds with the second level of the proposed description language. The steps can be `Serial` executions, `Parallel` executions, `GPU` executions or `I/O` executions. Each step is connected to the next one by a `Flow` arc, and the next step must be unique. The tasks start from the element without any input arc, and ends with the element without any output arc: a proper `Workload` specification includes only a single chain of elements. The time required from each step of the task is defined using their *Demand* properties. Parallel CPU executions require the corresponding demand for a number of threads that can be run in parallel as specified by the corresponding *NThreads* property. In the same way, GPU tasks splits into *NGPUThreads* parallel executions, all characterized by the same corresponding demand.

Systems described in this language, are analyzed by translating them in equivalent *multi-class fork/join queueing networks with finite capacity regions and class switching*. This type of systems can be analyzed using many available tools, such as JMT - Java Modeling Tool [30]. In particular, jobs are divided into classes and sub-classes. A class of jobs is associated to each `RepJob` and `Arrival` element. Closed classes are used for the former, and open classes for the latter. Parameter *N* of `RepJob` describes the population size of closed classes, and parameter *Rate* of `Arrival` defines the arrival rate of jobs. Moreover, if $Z \neq 0$, `RepJob` nodes generates also an infinite-server queue, with demand equal to the waiting time Z to account for the time each user elaborates the results of the previous iteration before sending a new job.

Each `Compute` node is transformed into up to three queues, representing respectively the CPU, the IO and the GPU of the corresponding infrastructure. If the considered node does not have a component (i.e. *NCores = 0*, *IOSpeed = 0* or *NGPUCores = 0*), the corresponding queue is not generated. The CPU is modeled by a c server processor sharing queue, where $c = NCores$ parameter of the corresponding `Compute` node. The I/O phase is modeled as a single server ordinary queue with First Come First Served discipline. The GPU is modeled by a c server processor sharing queue inside a *finite capacity region* (FCR). Parameter $c = NGPUCores$ models the parallelism of the

DOI: 10.1002/cpe

GPU, that is the number of cores it is composed of. Since the GPU is used in mutual exclusion among the competing processes, the FCR ensures that only one class of tasks is allowed to use resource at each time.

Sub-classes and class-switches with sub-class dependent routing are used to model the fact that the description of a workload can include more CPU, IO or GPU tasks. In particular, if more than one `Serial` or `Parallel` elements are included, the job is transformed into another sub-class and re-enters the same CPU queue. To account for the speed of the node, and for the length of the task, the demand of the considered sub-class is computed as the ratio of the *Demand* parameter of the task, and the *CoreSpeed* of the assiciated `Compute` node. If the speed of the node is load-dependent, then the law that defines the dependency is used to define a load-dependent demand for the class in the considered queue. Moreover, parallel tasks include a fork/join primitive that replicates the tasks as many times as defined by parameter *NThreads* and waits for all its components to finish before continuing to the next step. A similar approach is used for both the IO and the GPU. When a workload has been completed, the *Routing* property of the `Compute` node is used to route the job to the next station.

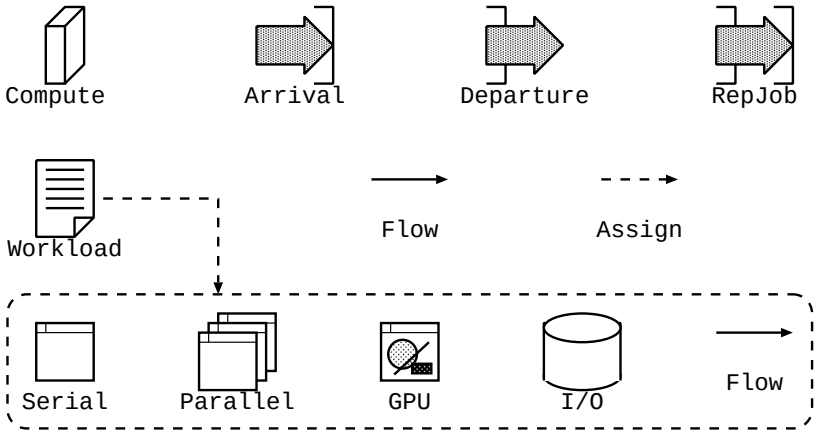

Figure 1. The tasks and system description language.

As an example, Figure 2a shows a single threaded application running in multicore environment. In this case, a workload (element `SingThread`) composed by a single-threaded CPU section (element `CPU`) and an I/O section (element `I/O`) is executed on a server (element `Server`) with a predefined number of cores and I/O speed. Since the considered workload does not have a GPU section, the corresponding queue is not included in the generated model. The workload is executed indefinitely, and starts again immediately after it is completed (element `Jobs`). The proposed model is automatically transformed into the queuing system shown in Fig. 2b. In particular, the equivalent queueing system is composed by three stations: one infinite server corresponding to the waiting the tasks experience before entering the system, one single server that represents the I/O component, and a multiple server that considers the CPU and the scheduler of the operating system. The multiplicity of the server of the queue corresponds to the number of cores of the CPU. More complex examples will be given in the rest of the paper.

5. VIRTUALIZED MULTICORE SYSTEMS

We now show the application of the modeling formalism to more complex scenarios by means of two models that respectively represent a virtualized multicore system without (in this Section) and with GPU (in the next Section), running single thread and multithread applications. In order to tune the models and validate them, a measurement campaign has been performed, part of the results of which, that are presented in this Section, have been already used in [1].

DOI: 10.1002/cpe

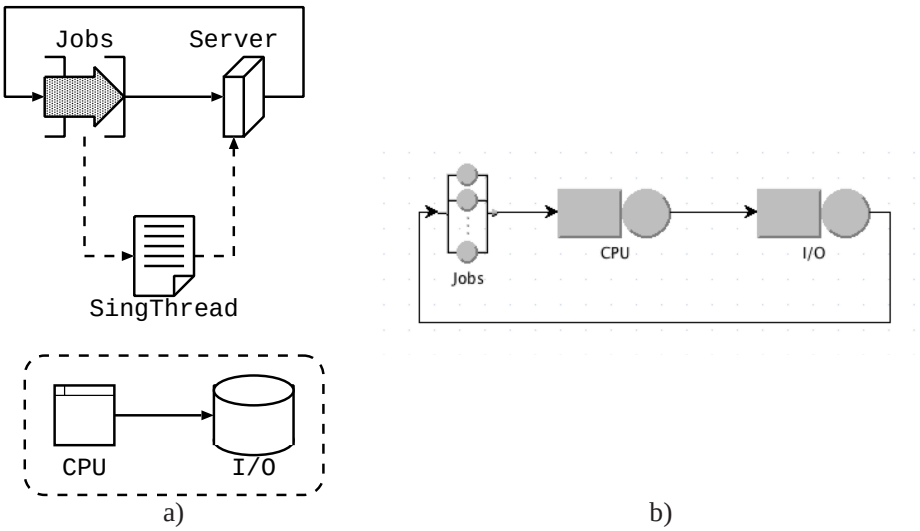

Figure 2. Model of a single-threaded application running on a multi-core system: a) using the proposed formalism, b) the automatically generated corresponding queueing model.

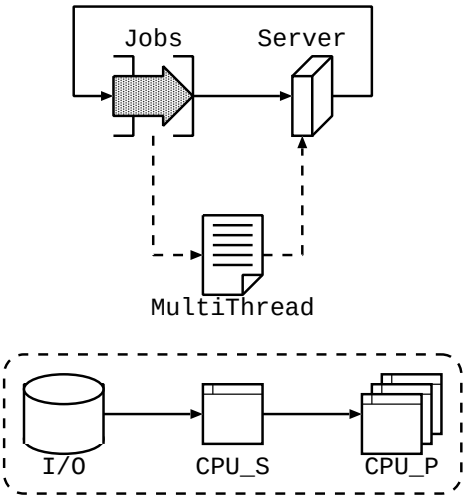

Figure 3. Model of a multi-threaded application running on a multi-core system.

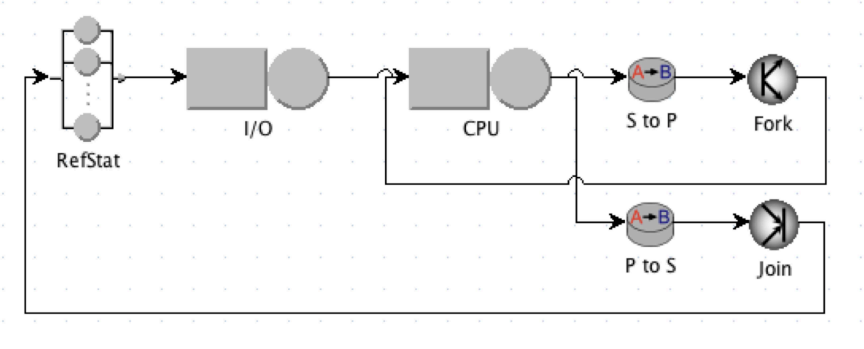

Figure 4. Model for multithreaded applications in multicore environments.

DOI: 10.1002/cpe

Element	Property	Description
`Compute`	*NCores*	Number of cores of the server
	CoreSpeed	Speed of the cores
	NGPUCores	Number of GPU cores of the server
	GPUCoreSpeed	Speed of the GPU cores
	IOSpeed	Speed of the I/O
	Routing	Routing policy of the jobs that have terminated their service
`Arrival`	*Rate*	Arrival rate for the corresponding workload
	Routing	Routing policy of the jobs entering the system
`Departure`	—	— *No parameters* —
`RepJob`	*N*	Number of jobs for the corresponding workload
	Z	Job waiting time
	Routing	Routing policy of the jobs starting their service
`Workload`	***	*Defined in the associated sub-model*
`Serial`	*Demand*	Average time spent in the serial part of the job
`Parallel`	*Demand*	Average time of each parallel part of the job
	NThreads	Number of threads of this parallel part of the job
`GPU`	*Demand*	Average time of each GPU task of the job
	NGPUThreads	Number of GPU threads of this part of the job
`I/O`	*Demand*	Average time spent in the I/O part of the job

Table I. Parameters of the formalism primitives

The measurement campaign aims to observe the behavior of the modeled platform running a benchmark, and show how the proposed models are able to describe it. The chosen benchmark is the DaCapo suite [31], from which the `batik` and `sunflow` benchmark applications have been chosen, respectively as mainly single threaded and multithread workload. `Batik` uses Apache Batik to generate SVG images[†], while `sunflow` is a 3D rendering application, based on parallel ray tracing algorithms. The running architecture is based on virtual machines running Linux OS being executed on Amazon EC2. Both applications are modeled as shown in Figure 3. Execution of the benchmark are driven by the `RepJob` element *Jobs*, and they are run on the node modeled by the `Compute` element *Server*. The workload is defined by a sub-model that includes an I/O phase (*I/O*), a serial CPU phase (*CPU_S*) and a parallel CPU phase (*CPU_P*). Of course, the `batik` benchmark will have a very short *CPU_P* phase, and long *CPU_S*. In contrast, `sunflow` will use a long *CPU_P* phase and a short *CPU_S*. The models are then automatically translated int the queueing network shown in Figure 4. In particular, the *Jobs* element is converted to the delay station *RefStat*. This node is also used as the reference station to compute the performance indices of the main job class. Since the workload does not require the use of the GPU, the *Server* node is mapped in two queues: *IO* and *CPU*. The single type of job is converted into two sub-classes: *S* to represent the serial execution of the benchmark, and *P* to model the parallel behavior. After the I/O execution, jobs first enters the CPU as class *S* jobs, then they switch to class *P* and are forked into as many jobs as threads. Each thread re-enters the CPU queue, to model its execution. All *P* jobs are then joined, and switched back to class *S* to repeat the workflow for the next request.

In the first experiment, the benchmarks are forced to use a single thread execution model (using a specific parameter of the DaCapo benchmarks). This is achieved by setting the property *NThreads* $= 1$ of the `CPU_P` element. The demand for the I/O is measured using the *iostat* Linux command. By considering the relations between CPU and I/O demand and the number of cores and I/O fraction, the experimental results are then fitted using Microsoft Excel to determine the model parameters,

[†]The transcoding of image elements may be performed by means of multiple threads, but the most of the processing is run by a single thread.

DOI: 10.1002/cpe

and the response time curve is estimated. CPU demands for computing (D_{CPU}) and I/O ($D_{I/O}$) are obtained by minimizing the squared distance between results from the model and the measured response times. Results show that accuracy for `sunflow` is good, with a mean error of 4.07% and a demand of 15.524, while for `batik` the error grows to 17.42% with a demand of 2.246. In fact, Figure 5 shows that the two approaches achieve very similar response times in all cases. This a consequence of the fact that this benchmark is highly parallel and essentially works in memory using large chunks of data, so that the contribution of the L2 shared cache provided by the architecture is minimal if not barely influential, as saturation of this resource happens almost immediately, due to the number and the volume of data transfers. The same choice instead is the cause of the significant error in the case of `batik`, the results of which are shown in Figure 6 for the considered cases. This is due to the fact that `batik` essentially processes data in a sequential task, and the effects of caching and of the presence of more cores, that are not completely exploited, are not negligible.

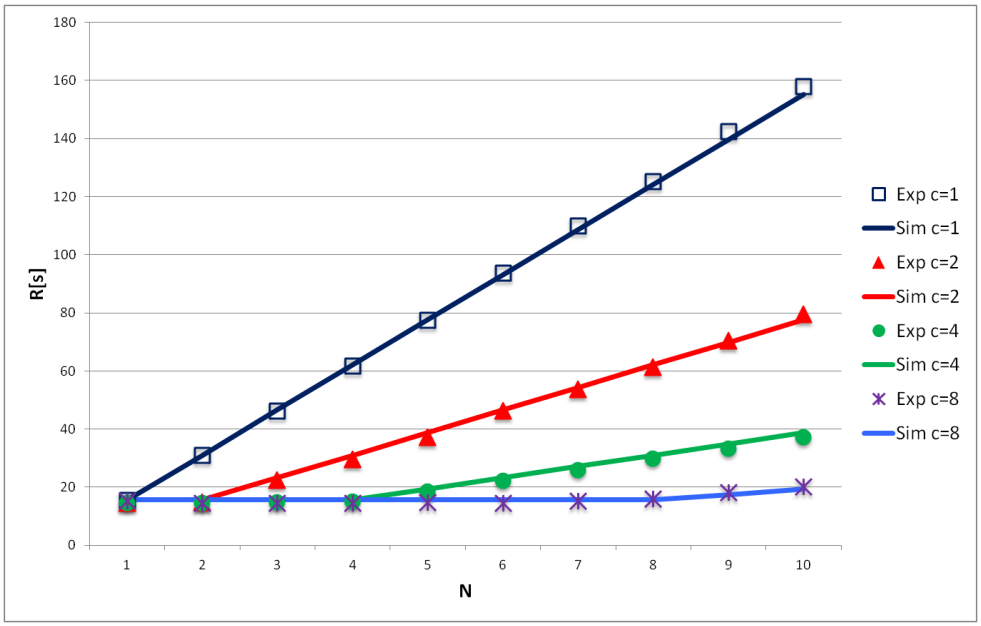

Figure 5. Model results and measurements for `sunflow`.

The problem can be solved by resorting to a load dependent approach for the CPU model, inspired to [32]. The parameter *CPUSpeed* of the `Compute` node *Server* (the service station that describes the CPU behavior) is made dependent on the workload.

By means of a new parameter fitting, load dependent server speed can be estimated so to account for the effect of the CPU architecture in absence of a complete saturation of all available cores. This technique allows the model to keep the constant service rate of the fully loaded system when there are enough threads to saturate all the cores, while compensating the other cases. The results obtained with the new approach for `batik` are in Figure 7, that shows a better fitness: the error in this case is now 4.53%, for a demand of 2.214. The `sunflow` case also benefits of the new approach, as the error is 0.85% and the demand is 15.704.

In the second experiment, the benefits of multithreading are analyzed. Multithreading allows a better exploitation of the various available cores since whenever the overall workload is not able to saturate the cores, more threads of the same application can actually run in parallel [33]. This has the effect of a reduction of the application execution time, that must be quantified in order to enable a correct modeling process.

DOI: 10.1002/cpe

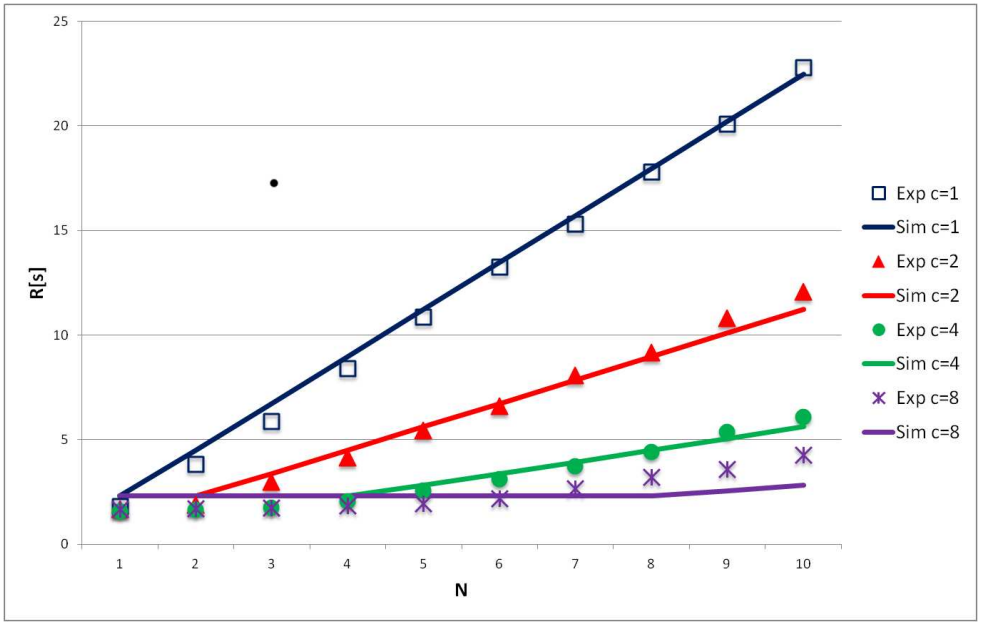

Figure 6. Model results and measurements for `batik`.

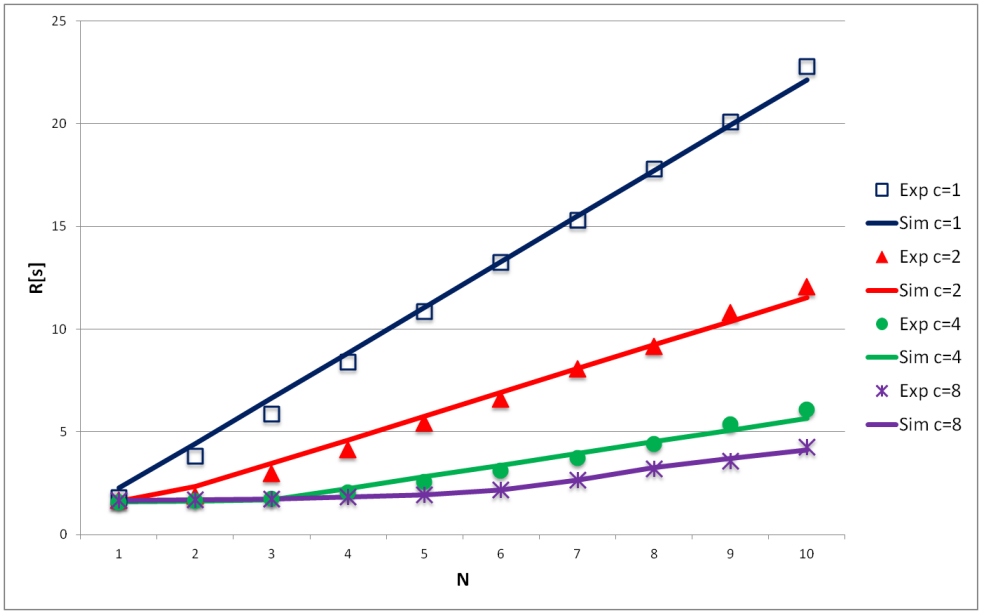

Figure 7. Comparison of results for the load-dependent model and measurements for `batik`.

`Sunflow` has been run on the reference architecture with different configurations, that enable an increasing number of threads[‡]. The related response times are shown in Figure 8 and Figure 9. As the nature of this benchmark is inherently highly parallelizable, due to the rendering algorithms used, the benefits are clearly visible in the figure whenever the number of jobs times the number of threads is less than the number of cores.

[‡]For the sake of completeness, the same analysis has been done on `batik` as well in [1].

DOI: 10.1002/cpe

In the model of Figure 3 multithreading is achieved by setting a value greater than one to the property *NThreads* of the `CPU_P` element. As before, the demands of the two phases can be fitted with respect to the measurements. Figure 8 and Figure 9 compares the response times predicted by the model with the ones measured on the reference architecture, a good agreement between the results can be observed.

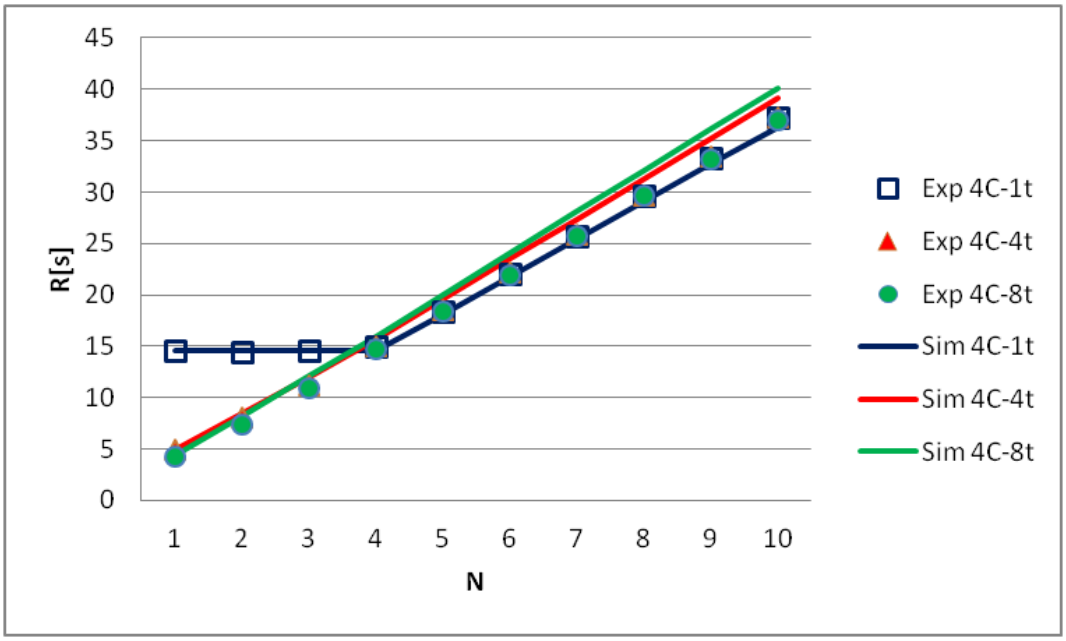

Figure 8. `sunflow` mean response time for four cores, and different threads and N.

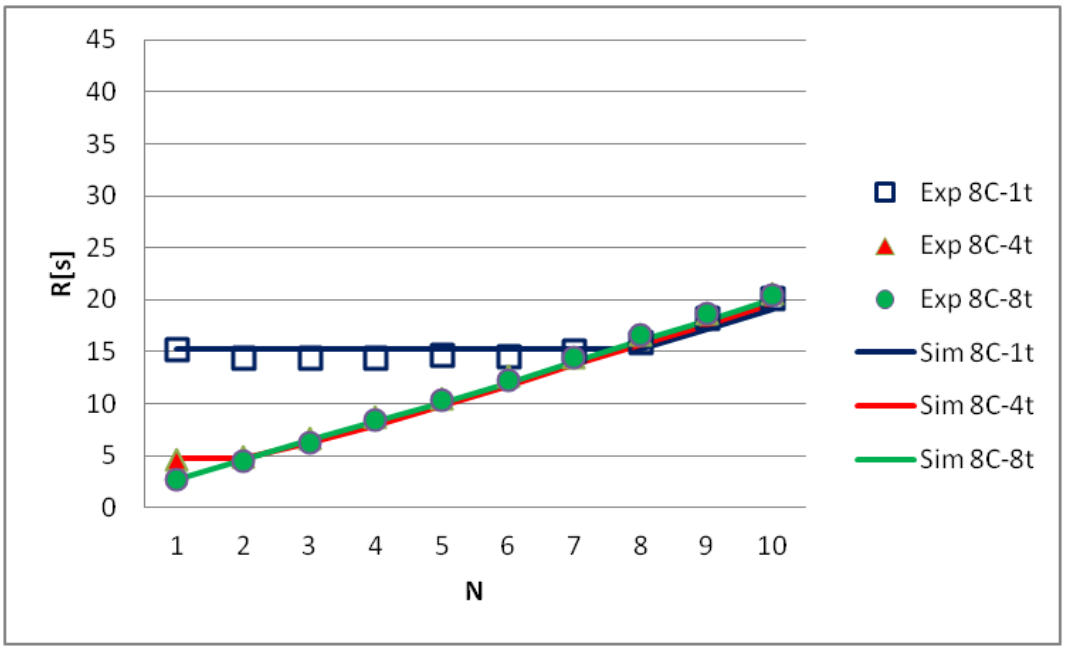

Figure 9. `sunflow` mean response time for eight cores, and different threads and N.

DOI: 10.1002/cpe

6. COMPLEX APPLICATIVE SCENARIO

In the last years, GPUs have been exploited to support the main CPU of a system in the execution of massively parallel applications. When the GPU is used to perform computation instead of graphics, applications are defined as *General-Purpose computing on Graphic Processing Unit* (GPGPU). Such applications are partitioned by the CPU in several independent tasks executed by threads which are allocated to specific GPU cores. When GPGPU applications are executed concurrently with CPU applications this approach introduces complex inter-dependencies among CPU and GPU that must be investigated in order to increase the global performance. In this section, we show how the proposed methodology can deal with this type of applications.

Fig. 10 shows an example of a model where two workloads are concurrently executed: a multi-threaded CPU application and a GPU intensive application. The multi-threaded application is similar to the one described in Figure 3. The GPU workload consists of a multi-threaded GPU section (element *GPU*) and a single-threaded CPU section (element *Serial*). The latter takes care of the allocation of threads to the GPU cores.

The model is automatically translated in the two-class queuing network system shown in Fig. 11. The two classes represent the CPU and GPU intensive workloads, respectively. According to their class, the jobs follow a different path: CPU intensive jobs starts from the delay station *CPU_Jobs*, then they perform I/O operations in the *I/O* queue, finally enter the CPU (station *CPU*). When they exit from the *CPU*, jobs change their sub-class from "S" (Serial) to "P" (Parallel), and experience a fork (node *CPU-thread*) to simulate the split into several parallel jobs. Parallel jobs are routed again to the CPU, then they are merged back into a single element in the join node *CPU-Thread-end*, and return to the "P" sub-class before finishing and returning to the *CPU_Jobs* node. Instead, GPU intensive jobs are forwarded by the delay station *GPU_Jobs* directly to the *CPU* queue, thus avoiding the I/O service. After the CPU service, GPU class jobs are routed to the sub-system composed by the *GPU-thread* fork node, the *GPU* queue in the corresponding finite capacity region, and finally joined back in the *GPU-thread-end* join node. In this case, the maximum number of threads is limited to the number of GPU cores by the FCR. The model is then characterized by a total of 7 time durations: delay for both the CPU and the GPU applications, serial part of the GPU application, time for each GPU thread, I/O duration for the CPU application, serial and parallel duration of the CPU application.

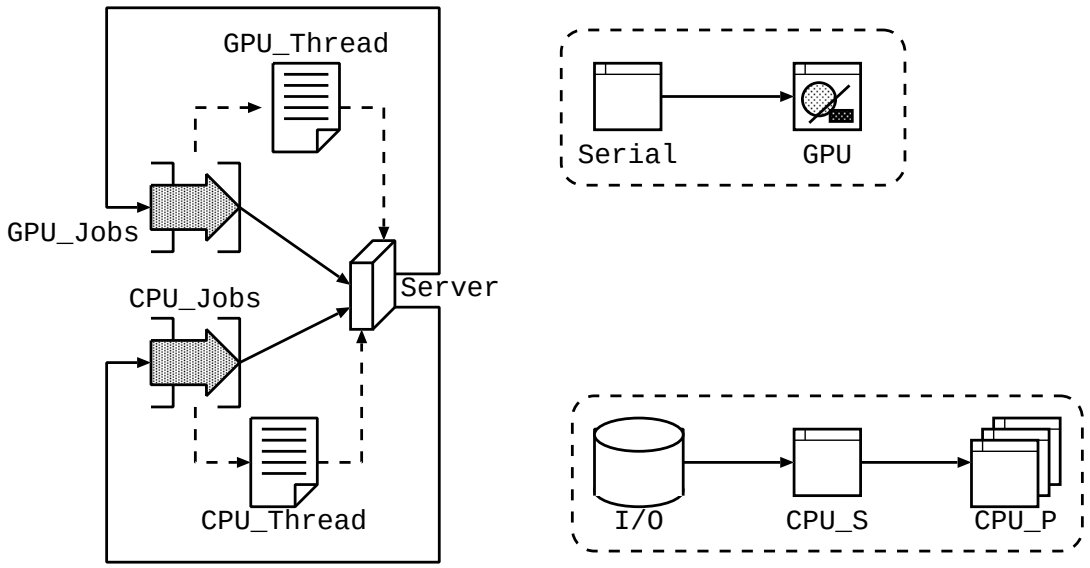

Figure 10. Model of a system running two different types of application, with one characterized by GPGPU workload.

DOI: 10.1002/cpe

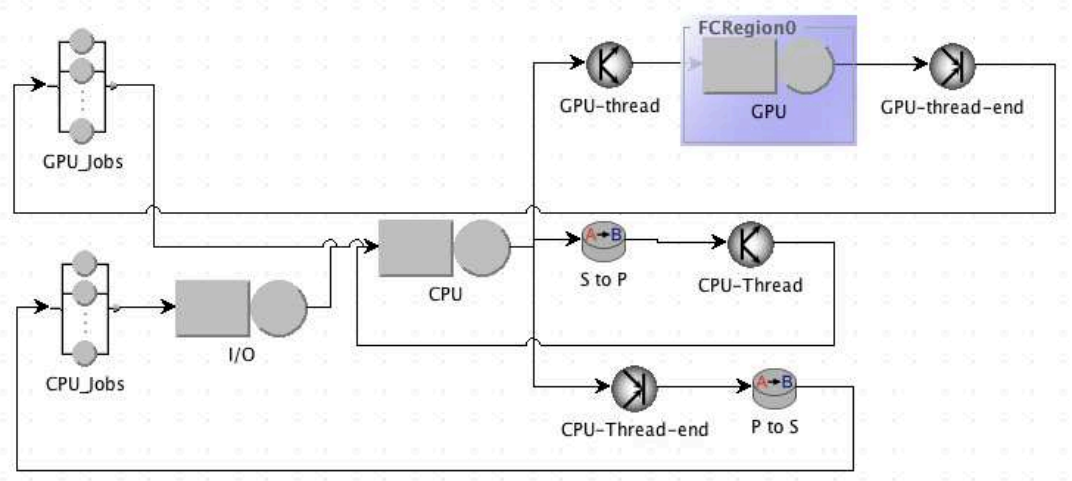

Figure 11. Queuing network of a system running two different types of application, with one characterized by GPGPU workload.

To validate the model against real system measurement, we perform the following experiments using a desktop PC with an i7-3770 CPU@3.4GHz. with eight cores and 16 GB. The GPU is a GeForce GTX 560 with 384 cores. In all experiments we set the maximum number of CPU (GPU) threads equal to the number of CPU (GPU) cores. Initially we analyze the model with just a GPU workload (i.e. a single GPU class model) and compare the results with a GPU benchmark. We consider a CUDA [34] implementation of the eigenvalues computation of a square matrix. In such algorithm, the original problem is hierarchically decomposed in several sub-problems which can be solved separately by different threads allocated to the GPU cores, thus resulting in an high-parallel GPU application. With a single class model the number of parameters needed to characterize the system is reduced to 4: users delay, I/O duration, CPU serial part and GPU parallel part.

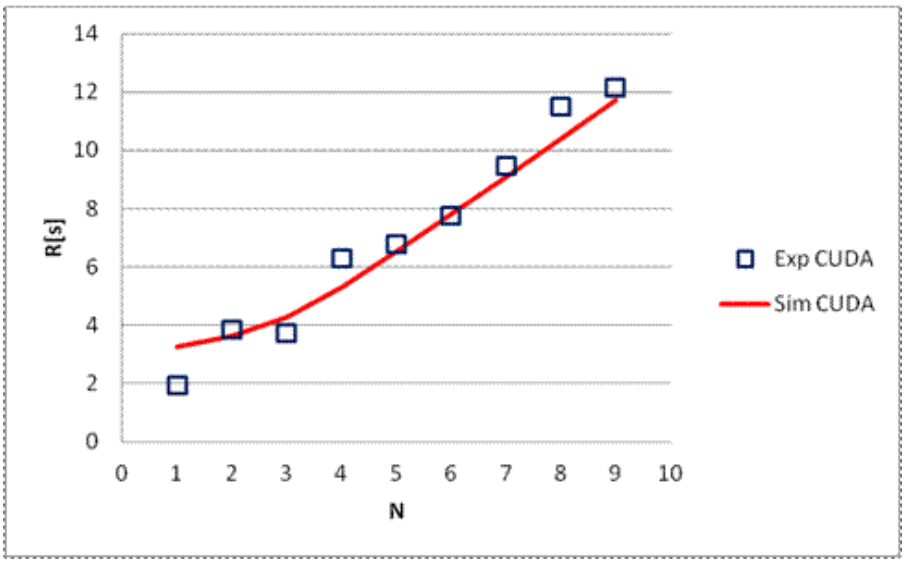

Figure 12. CUDA mean response time for different values of N.

In order to determine the parameter values, we use a Nelder-Mead simplex algorithm [35] to find the minimum of a multivariate function, i.e. the square error between the computed system response time and the experimental one, thus performing a non linear least square minimization.

DOI: 10.1002/cpe

The Nelder-Mead simplex algorithm is implemented in the *scipy.optimize* module [36] of Python language. Even with this simple algorithm, we obtain quite satisfactory results with a mean relative error $\sigma = 11.23\%$. A graphical comparison between simulated and experimental system response time as a function of the number of jobs N is shown in Fig. 12.

Then we analize the two-class model with both CPU and GPU intensive workloads. Let N_{CPU} and N_{GPU} be the number of CPU and GPU intensive jobs, respectively. Since the model is closed the total number of jobs N is constant with $N = N_{CPU} + N_{GPU}$. In a two-class model the average time needed to complete a job of a specific class c (also called per-class response times) depends on the proportion of class c jobs inside the system. We define the population mix of the CPU intensive jobs as $\beta = N_{CPU}/N$ and investigate the model for values $\beta \in [0.1, 0.9]$ with step-size 0.1.

We consider two multi-class scenarios: in both of them the GPU workload is given by the CUDA eigenvalues computation, whereas the CPU workload is given by either `sunflow` or `batik` benchmark, respectively. As before, to parametrize the model we perform a fitting procedure based on the Nelder-Mead simplex algorithm when this time we minimize the sum of the square errors between the computed and experimental system per-class response times for all values of β. In such a way, we aim to obtain a single set of parameter values that fit independently of β.

The graphic comparison between simulated and experimental per-class response times as a function of the population mix β is shown in Fig. 13 for the CUDA-Sunflow scenario, and in Fig. 14 for the CUDA-Batik scenario. The mean relative error in the first scenario is $\sigma = 22.02\%$, in the second one is $\sigma = 10.91\%$.

In this case the worst model accuracy is obtained in the CUDA-Sunflow scenario where both per-class response times are not properly fitted. Instead in the CUDA-batik scenario at least the batik workload is well fitted. It appears that the interactions between CUDA and Sunflow workloads are not well captured by the model, a different fitting procedure would lead to a better agreement between the model and the real data. We decided to leave this investigation outside the scope of this work.

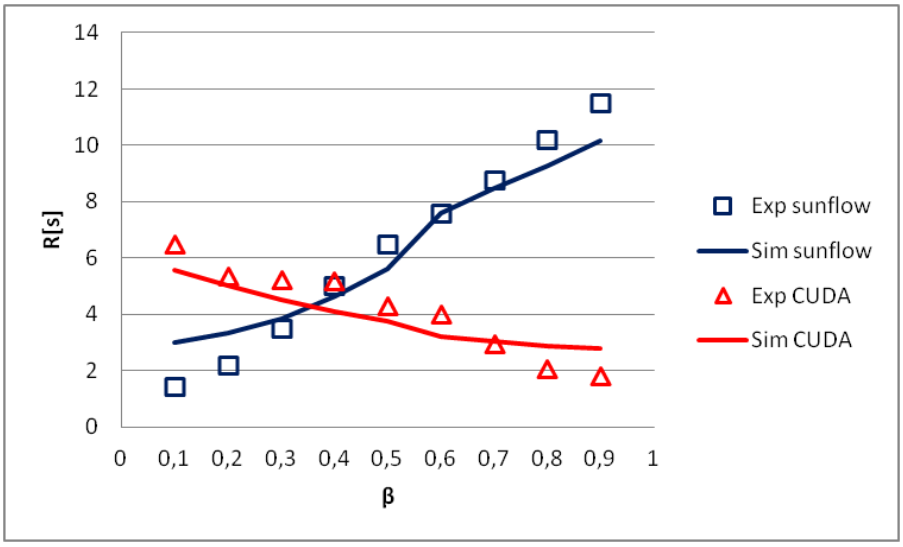

Figure 13. `Sunflow` and `CUDA` mean response time for different values of N.

7. CONCLUSIONS

In this paper we proposed a modeling language and a technique for the development and the tuning of models for the performance evaluation of multithreaded applications in multicore environments with GPU support. The proposed technique requires a limited number of parameters to characterize

DOI: 10.1002/cpe

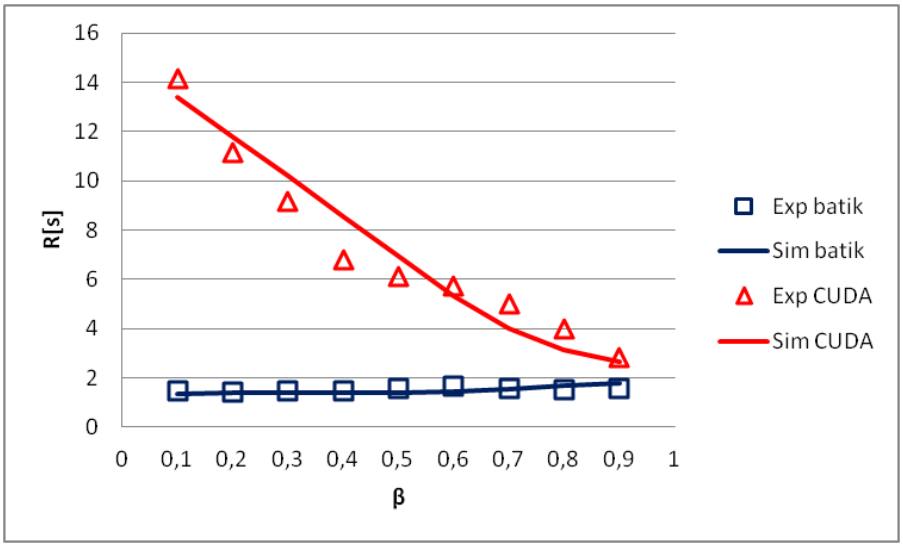

Figure 14. `Batik` and `CUDA` mean response time for different values of N.

the models. Some examples of model tuning have been provided, by means of proper measurements of real benchmark applications on multicore and GPU machines. The proposed results are intended to support the design process of complex Big Data applications and cloud infrastructures, in conjunction to prior authors' work, and give a foundation to a general approach to performance design and assessment in this field.

REFERENCES

1. Cerotti D, Gribaudo M, Iacono M, Piazzolla P. Workload characterization of multithreaded applications on multicore architectures. *ECMS*, European Council for Modeling and Simulation, 2014; 480–486.
2. Iacono M, Barbierato E, Gribaudo M. The SIMTHESys multiformalism modeling framework. *Computers and Mathematics with Applications* 2012; (64):3828–3839, doi:10.1016/j.camwa.2012.03.009.
3. Barbierato E, Gribaudo M, Iacono M. Defining Formalisms for Performance Evaluation With SIMTHESys. *Electr. Notes Theor. Comput. Sci.* 2011; **275**:37–51.
4. Vittorini V, Iacono M, Mazzocca N, Franceschinis G. The OsMoSys approach to multi-formalism modeling of systems. *Software and System Modeling* 2004; **3**(1):68–81.
5. Franceschinis G, Gribaudo M, Iacono M, Marrone S, Moscato F, Vittorini V. Interfaces and binding in component based development of formal models. *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS '09, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering): ICST, Brussels, Belgium, Belgium, 2009; 44:1–44:10.
6. Gaonkar S, Keefe K, Lamprecht R, Rozier E, Kemper P, Sanders WH. Performance and dependability modeling with möbius. *SIGMETRICS Perform. Eval. Rev.* Mar 2009; **36**(4):16–21, doi:10.1145/1530873.1530878. URL `http://doi.acm.org/10.1145/1530873.1530878`.
7. Barbierato E, Rossi GLD, Gribaudo M, Iacono M, Marin A. Exploiting product forms solution techniques in multiformalism modeling. *Electronic Notes in Theoretical Computer Science* 2013; **296**(0):61 – 77, doi:http://dx.doi.org/10.1016/j.entcs.2013.07.005. URL `http://www.sciencedirect.com/science/article/pii/S1571066113000364`.
8. Barbierato E, Gribaudo M, Iacono M, Marrone S. Performability modeling of exceptions-aware systems in multiformalism tools. *ASMTA*, 2011; 257–272.
9. Barbierato E, Gribaudo M, Iacono M. A performance modeling language for big data architectures. *ECMS*, Rekdalsbakken W, Bye RT, Zhang H (eds.), European Council for Modeling and Simulation, 2013; 511–517. URL `http://dblp.uni-trier.de/db/conf/ecms/ecms2013.html#BarbieratoGI13`.
10. Barbierato E, Gribaudo M, Iacono M. Modeling apache hive based applications in big data architectures. *Proceedings of the 7th International Conference on Performance Evaluation Methodologies and Tools*, ValueTools '13, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering): ICST, Brussels, Belgium, Belgium, 2013; 30–38, doi:10.4108/icst.valuetools.2013.254398. URL `http://dx.doi.org/10.4108/icst.valuetools.2013.254398`.
11. Castiglione A, Gribaudo M, Iacono M, Palmieri F. Exploiting mean field analysis to model performances of Big Data architectures. *Future Generation Computer Systems* 2013; (0):–, doi:http://dx.doi.org/10.1016/j.future.2013.07.016.

12. Barbierato E, Gribaudo M, Iacono M. Performance evaluation of nosql big-data applications using multi-formalism models. *Future Generation Computer Systems* 2013; **to appear**, doi:http://dx.doi.org/10.1016/j.future.2013.12.036.
13. Castiglione A, Gribaudo M, Iacono M, Palmieri F. Modeling performances of concurrent big data applications. *Software: Practice and Experience* 2014; :n/a–n/adoi:10.1002/spe.2269. URL http://dx.doi.org/10.1002/spe.2269.
14. Qureshi MK, Patt YN. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, IEEE Computer Society: Washington, DC, USA, 2006; 423–432.
15. Kavadias SG, Katevenis MG, Zampetakis M, Nikolopoulos DS. On-chip communication and synchronization mechanisms with cache-integrated network interfaces. *Proceedings of the 7th ACM international conference on Computing frontiers*, CF '10, ACM: New York, NY, USA, 2010; 217–226.
16. Liu F, Jiang X, Solihin Y. Understanding how off-chip memory bandwidth partitioning in chip multiprocessors affects system performance. *HPCA 2010*, Jan.; 1–12.
17. Schneider S, Yeom JS, Nikolopoulos D. Programming multiprocessors with explicitly managed memory hierarchies. *Computer* Dec; **42**(12):28–34.
18. Moseley T, Kihm J, Connors D, Grunwald D. Methods for modeling resource contention on simultaneous multithreading processors. *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on*, Oct.; 373–380.
19. Kim Y, Han D, Mutlu O, Harchol-Balter M. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. *HPCA 2010*, Jan.; 1–12.
20. Menasce' DA. Virtualization: Concepts, applications, and performance modeling. *Proc. of The Computer Measurement Groups 2005 International Conference*, 2005.
21. Benevenuto F, Fernandes C, Santos M, Almeida VAF, Almeida JM, Janakiraman GJ, Santos JR. Performance models for virtualized applications. *ISPA Workshops*, *Lecture Notes in Computer Science*, vol. 4331, Min G, Martino BD, Yang LT, Guo M, Rnger G (eds.), Springer, 2006; 427–439.
22. Huber N, Von Quast M, Brosig F, Kounev S. Analysis of the performance-influencing factors of virtualization platforms. *Proceedings of the 2010 international conference on On the move to meaningful internet systems: Part II*, OTM'10, Springer-Verlag: Berlin, Heidelberg, 2010; 811–828.
23. Cherkasova L, Gardner R. Measuring cpu overhead for i/o processing in the xen virtual machine monitor. *Proc. of the USENIX Annual Technical Conference*, ATEC '05, USENIX Association: Berkeley, CA, USA, 2005; 24–24.
24. Che S, Boyer M, Meng J, Tarjan D, Sheaffer JW, Skadron K. A performance study of general-purpose applications on graphics processors using cuda. *J. Parallel Distrib. Comput.* Oct 2008; **68**(10):1370–1380, doi:10.1016/j.jpdc.2008.05.014. URL http://dx.doi.org/10.1016/j.jpdc.2008.05.014.
25. Owens J, Houston M, Luebke D, Green S, Stone J, Phillips J. Gpu computing. *Proceedings of the IEEE* May 2008; **96**(5):879–899, doi:10.1109/JPROC.2008.917757.
26. Gregg C, Hazelwood K. Where is the data? why you cannot debate cpu vs. gpu performance without the answer. *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, 2011; 134–144, doi:10.1109/ISPASS.2011.5762730.
27. Shi L, Chen H, Sun J, Li K. vcuda: Gpu-accelerated high-performance computing in virtual machines. *Computers, IEEE Transactions on* June 2012; **61**(6):804–816, doi:10.1109/TC.2011.112.
28. Yeh CY, Kao CY, Hung WS, Lin CC, Liu P, Wu JJ, Liu KC. Gpu virtualization support in cloud system. *Grid and Pervasive Computing*, *Lecture Notes in Computer Science*, vol. 7861, Park J, Arabnia H, Kim C, Shi W, Gil JM (eds.). Springer Berlin Heidelberg, 2013; 423–432, doi:10.1007/978-3-642-38027-3_45. URL http://dx.doi.org/10.1007/978-3-642-38027-3_45.
29. Gribaudo M, Piazzolla P, Serazzi G. Consolidation and replication of vms matching performance objectives. *Analytical and Stochastic Modeling Techniques and Applications*, *Lecture Notes in Computer Science*, vol. 7314. Springer Berlin Heidelberg, 2012; 106–120.
30. Bertoli M, Casale G, Serazzi G. Jmt: performance engineering tools for system modeling. *SIGMETRICS Perform. Eval. Rev.* 2009; **36**(4):10–15, doi:http://doi.acm.org/10.1145/1530873.1530877.
31. Blackburn SM, Garner R, Hoffmann C, Khang AM, McKinley KS, Bentzur R, Diwan A, Feinberg D, Frampton D, Guyer SZ, *et al.*. The dacapo benchmarks: Java benchmarking development and analysis. *SIGPLAN Not.* Oct 2006; **41**(10):169–190, doi:10.1145/1167515.1167488. URL http://doi.acm.org/10.1145/1167515.1167488.
32. Cerotti D, Gribaudo M, Piazzolla P, Serazzi G. Flexible cpu provisioning in clouds: A new source of performance unpredictability. *QEST*, 2012; 230–237.
33. Cerotti D, Piazzolla P, Gribaudo M, Serazzi G. End-to-end performance of multi-core systems in cloud environments. *EPEW*, 2013; 221–235.
34. CUDA Parallel Programming Language Website. http://www.nvidia.com/object/cuda_home_new.html.
35. Nelder JA, Mead R. A simplex method for function minimization. *The Computer Journal* Jan 1965; **7**(4):308–313, doi:10.1093/comjnl/7.4.308. URL http://dx.doi.org/10.1093/comjnl/7.4.308.
36. SciPy Website. http://www.scipy.org/scipylib/index.html.