UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E INFORMÁTICA INDUSTRIAL

HUDSON JORGE

EXTENSIONS TO CLOUDSIM SIMULATOR BASED ON A FIRST ORDER PARASITIC COUPLING OVERHEAD MODEL TO STUDY THE CPU SCALATION IN CLOUDS SYSTEMS

DISSERTAÇÃO

CURITIBA

2018

HUDSON JORGE

EXTENSIONS TO CLOUDSIM SIMULATOR BASED ON A FIRST ORDER PARASITIC COUPLING OVERHEAD MODEL TO STUDY THE CPU SCALATION IN CLOUDS SYSTEMS

Dissertação apresentada ao Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial da Universidade Tecnológica Federal do Paraná como requisito parcial para obtenção do grau de "Mestre em Ciências" – Área de Concentração: Informática Industrial.

Orientador: Emilio C. G. Wille

CURITIBA

2018

Dados Internacionais de Catalogação na Publicação

Jorge, Hudson
Extensions to cloudsim simulator based on a first order parasitic coupling overhead model to study the CPU scalation in clouds systems [recurso eletrônico] / Hudson Jorge.-- 2019.
1 arquivo eletrônico (74 f.): PDF; 1,49 MB.

Modo de acesso: World Wide Web
Título extraído da tela de título (visualizado em 22 abr. 2019)
Texto em inglês com resumo em português
Dissertação (Mestrado) - Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial, Curitiba, 2018
Bibliografia: f. 72-74.

1. Engenharia elétrica - Dissertações. 2. Processamento paralelo (Computadores). 3. Processamento paralelo (Computadores) - Simulação por computador. 4. Computadores - Redes. 5. Modelagem de sistemas complexos. 6. Dispositivos de treinamento simulado. I. Wille, Emílio Carlos Gomes. II. Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial. III. Título.

CDD: Ed. 23 – 621.3

Biblioteca Central da UTFPR, Câmpus Curitiba.
Bibliotecário: Adriano Lopes CRB-9/1429

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Diretoria de Pesquisa e Pós-Graduação

TERMO DE APROVAÇÃO DE DISSERTAÇÃO Nº 810

A Dissertação de Mestrado intitulada **"*Extension to Cloudism Simulator Based on a First Order Parasitic Coupling Overhead Model to Study the CPU Scalation in Cloud Systems*"** defendida em sessão pública pelo(a) candidato(a) **Hudson Jorge**, no dia 22 de outubro de 2018, foi julgada para a obtenção do título de Mestre em Ciências, área de concentração Telecomunicações e Redes, e aprovada em sua forma final, pelo Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial.

BANCA EXAMINADORA:

Prof(a). Dr(a). Emilio Carlos Gomes Wille - Presidente – (UTFPR)
Prof(a). Dr(a). Luis Carlos Erpen de Bona - (UFPR)
Prof(a). Dr(a). Elias Procópio Duarte - (UFPR)

A via original deste documento encontra-se arquivada na Secretaria do Programa, contendo a assinatura da Coordenação após a entrega da versão corrigida do trabalho.

Curitiba, 22 de outubro de 2018.

To my parents, that gave all they had, material and spiritual, to support my ambitions and achievements. My efforts are to honor them enough, even knowing it is an impossible task.

To my daughter and son, for the opportunity to give them what my parents gave to me.

To my friends, for the good advices and the knowledge shared.

To my mentor, for the patience and professionalism.

AGRADECIMENTOS

There are always a set of people that without them it would be impossible to achieve our projects and ambitions. There is not enough words to thanks all of them for all they gave in terms of support, knowledge, advice or just acceptance. But some are crucial in the long run projects, and this work is one of them.

Special thanks for my mentor Prof. Dr. Emilio C. G. Wille, for the patience in listening hours of contradictory explanations, correcting me and always putting on focus the target. Special thanks for my family, that supported long periods of absent, even when I was with them. I wish to thanks my business coleagues that helped me when I needed knowledge they have and shared gracefully.

In memory of Munir Jorge, my beloved Father.

RESUMO

JORGE, Hudson. EXTENSIONS TO CLOUDSIM SIMULATOR BASED ON A FIRST ORDER PARASITIC COUPLING OVERHEAD MODEL TO STUDY THE CPU SCALATION IN CLOUDS SYSTEMS. 74 f. Dissertação – Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial, Universidade Tecnológica Federal do Paraná. Curitiba, 2018.

O aumento de performance em CPUs paralelas é dependente de muitos fatores como comportamento das aplicações, cache disponível, eficiência do uso do cache. Simuladores em computadores são usados para prever ou comparar a performance de processadores multi-core; eles são categorizados como modelos ciclo-precisos ou de alto nível. Os modelos cliclo-precisos são baseados na teoria de filas para passo-a-passo conduzir a simulação a executar as operações necessárias para consumir um modelo que representa o tamanho da aplicação. Os modelos de alto nível são usados para coletar, usualmente através de hardware adicional, informações não acessíveis que representam o comportamento do sistema em operação. Nos dois casos, eles são complexos porque uma longa sequência de filas (ciclo-preciso) controlam o processo, fazendo com que a simulação sejam muito demoradas. No caso de alto nível, muitas vezes é necessário um hardware completamente diferente do sistema real a fim de ler os estados necessários. Além disso, às vezes sistemas com lógica programável são usados para implementar o controle de cache, filas e protocolos de comunicação o que aumenta o tempo de execução, por causa as restrições de clock desse hardware adicional. Baseado em um modelo simples proposto por Kandalintsev e Lo Cigno, o Modelo BFO (First Order Performance), toda a complexidade da arquitetura interna é estudada como interferência parasítica de uma CPU/core lutando por recursos comuns. Eles introduziram um fator de acoplamento, que incorpora a essencia do comportamento de performance observado em um sistema. Neste trabalho, usamos o modelo BFO para implementar um simulador ciclo-preciso rápido capaz de manusear grandes sistemas. Para tanto o simulador CloudSim foi extendido para se comportar como um simulador ciclo-preciso. Basicamente implementamos modelos de CPU e modelos de aplicação que poder dar conta da complexidade de diferentes tipos de cenários (comportamento de CPU distintos quando rodando diferentes tipos de aplicação). Usando medidas off line nós treinamos o modelo para extrair os fatores de acoplamento necessários para alimentar o simulador. Testamos o comportamento em pares para estar seguros de que a implementação poderia reproduzir os testes medidos. Nossas simulações consideram até 14 cores (no caso de m Xeon E5-2880v4) e para melhorar a qualidade dos resultados introduzimos um fator de correção para minimizar o erro observado (que parece ser exponencial), com bons resultados.

Palavras-chave: processamento paralelo, arquiteturas multi-core, computação de alta performance, technologia, simulação, modelamento de performance.

ABSTRACT

JORGE, Hudson. EXTENSION TO CLOUDSIM SIMULATOR BASED ON A FIRST ORDER PARASITIC COUPLING OVERHEAD MODEL TO STUDY THE CPU SCALATION IN CLOUDS SYSTEMS. 74 f. Dissertação – Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial, Universidade Tecnológica Federal do Paraná. Curitiba, 2018.

Parallel CPU performance scaling is dependent of many factors such applications behavior, cache available, efficiency of cache use. Computer simulators are used to predict or benchmark the performance of multi-core processors; they are categorized as cycle accurate or high-level models. The cycle accurate models are based on queue theory to step-by-step conduct the simulation to perform the operations needed to consume a model that represents the application size. The high-level models are used to collect, usually via additional hardware, not accessible information that represents the behavior of the running system. In those two cases, they are complex because a very long sequence of queues (cycle accurate) controls the process, making the simulation very time consuming. And in the case of high-level, sometimes it needs a complete different hardware from the actual system to read the needed states. In addition, sometimes programmable logic systems are used to implement the cache controls, queues and communication protocols what enlarges the running time, to accommodate the clock restrictions of the additional hardware. Based on a simple model proposed by Kandalintsev and Lo Cigno, the Behavioral First Order Performance Model (BFO), all the complexity of the internals of the architecture is studied as parasitic interference of one CPU/core fighting for shared resources. They introduced a coupling factor, which incorporates the essence of the observed performance behavior in a system. In this work, we used the BFO model to implement a fast cycle-accurate simulator capable of handling even large systems. To do it the CloudSim simulator was extended to behave as a cycle-accurate simulator. Basically we implemented CPU and application models that can handle the complexity of the different types of scenarios (distinct CPU behavior when running different kinds of applications). Using off line measurements we trained the model to extract the coupling factor needed to feed the simulator. We tested the pairwise behavior to make sure that the implementation could reproduce the measured tests. Our simulations consider up to 14 cores (in case of Xeon E5-2880v4) and to improve the results quality we introduced a correction factor to minimize the (seemly exponential) error observed, with good results.

Keywords: parallel processing, multi-core architectures, high performance computing, technology, simulation, performance modeling.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

HPC	High Performance Computing
BCE	Base Core Equivalent
SLATE	System-Level Analysis Tool for Early Exploration
BCE	Base Core Equivalent
FPGA	Field Programmable Gate Array
API	Application Programming Interface
DVFS	Dynamic Voltage and Frequency Scaling
MIPS	Million Instructions Per Second
GPU	Graphics Processing Unit
SSD	Solid State Disk

SUMÁRIO

1 INTRODUCTION

One well known empiric result in HPC (High Performance Computing) area is that *n* identical CPUs working in parallel will not deliver *n* times the speed of one CPU alone, independent of the architecture used for interconnect them. There are many reasons for that, for example memory and cache availability, I/O interruptions and disk wait time. This measurable behavior points towards that no multi-CPUs architecture can give exclusive resources to each individual CPU, making the overall performance decreases, due to the shared resources competition among the CPUs. In addition to it, the software also play an important role since it should be designed to make use of this parallel CPUs environment, implementing algorithms optimized for this parallel architecture.

This has been a long time discussion, that probably started with the famous observation of Amdahl (AMDAHL, 1967) that statistically a software execution in a multi-CPU environment can be divided in two parts, the parallel ratio that represents the fraction of the execution time where the CPUs have all the resources (physical and logical) available to execute their jobs and the serial ratio that represents the fraction of the execution time where the CPUs have to wait for a resource be free or available. In the time of Amdahl's observation the serial fraction was estimated to be as big as 30%.

Overtime, new architectures came to better supply the CPUs with promptly available resources, it was pushed by the needs of the supercomputation area, these new architectures along with optimized software algorithms pushed the HPC area to new standards. Gustafson (GUSTAFSON, 1988) noticed that the Amdahl's law was not predicting well the measured performance in his lab. The key assumption he stated is that Amdahl's law was too restrictive because assumed that the serial fraction remained the same size independent of the problem size to be computationally solved, that is why nowadays the Amdahl's law is also known as *Fixed Size* law. According to Gustafson, the software designers make use of the new multi-CPU architecture resources and developed new algorithms that escalates down the serial fraction in

the same proportion of the problem size. This means that if the problem is doubled in size, the serial fraction is halved. This is known as Gustafson's law or *Fixed Time* law. It is *Fixed Time* because the serial fraction has the same absolute time independent of the problem size.

This discussion, was pacified for some time in HPC area, because the *Fixed Time* law assured that the multi-CPU architectures could perform very well if the software (Operational System (OS) and Algorithms) was designed to extract the best of these architectures. However, in the domestic and general purpose computation area the single CPU architecture was the common approach, since the Moore's law (MOORE, 1965) was still delivering results. In the servers market, very limited multi-core CPUs architectures were available for use, mainly because of the costs to implement these chips. Sudenly it all changed, mainly because of the Moore's law failings (TRACK et al., 2017), the multi-core approach presented as the only viable solution to continue the exponential growth of the power computing.

Since than, the discussion of CPU performance scaling reopenned, the main driver is because, even if the multi-core CPUs implementations are based on HPC architectures, they are more limited in a silicon chip because of restricted available space and limitations of the interconnection topologies possible to be implemented between the processors. For example some supercomputers implement hypercubic interconexion topology, what is hard to implement in silicon. Based on these limitations Marty and Hill (HILL; MARTY, 2008) have developed a corolary of the Ahmdahl's law, based on the concept of BCE (Base Core Equivalent). This work served as a baseline for Sun and Ni's work (SUN; NI, 1990), they created a model adherent to multi-core processor architectures, showing that they can perform near to Gustafson's law if there is enough cache memory available inside the chip. Moreover, Sun and Chen (SUN; CHEN, 2010) showed that it is possible to perform reasonably well even in presence of the *Memory Wall* problem (that is the disparity between how fast a CPU can operate on data and how fast it can get/put data on memory), but the effort is to be put in faster and larger cache memories and algorithms to clean it from leftover garbage.

The statistical models are fundamental to guide the research and industry on where to put efforts to improve performance over the generations of multi-core processors, as well as guide the software research to improve the use of the parallel architecture. However, they cannot predict the actual performance, also known as speed up, of a system based on the architecture and the software currently running on top of it. Therefore, there is room for other

kind of performance models, one capable to predict the performance in the systems based on observable statistics collected on runtime. There are several works in this area, usually they are or too complex for implementation in real systems or are based on “hard to measure” parameters. For example, Barroso et. al. (BARROSO et al., 1995) created the RPM (Rapid Prototyping for Multiprocessor) an engine based on queue analysis, it presents a very good precision on the predictions, but it is slow to run since it executes the pipe of instructions in dedicated hardware (like cache and cache controller) to access the needed cache statistics. Nanda et. al. (NANDA et al., 2000) developed MemorIES that substitute physically the L1, L2 and L3 caches by a dedicated hardware capable to collect the cache statistics and calculate the processor’s performance. This model is used to orient the design of servers and besides being faster then other models it uses parameters inaccessible in commercial servers.

To address the need for a model capable of deliver good and fast results based on observable statistics (in multi-core processors, usually the HPC counters) Kandalintsev and Lo Cigno (LO CIGNO; KANDALINTSEV, 2012) proposed a simple model based on the measured behavior an application presents when running on the multi-core system in controlled conditions and corrects it for the normal conditions of use. This model is further developed in Kandalintsev PhD thesis (KANDALINTSEV, 2016) *”Application Interference in Multi-Core Architectures: Analysis and Effects”* where a methodology to collect the parameters of the model are described as well as the methodology to test the model against the real world systems, even a methodology to collect the statistics to feed online algorithms, specially in case of CRM (Cloud Resource Manager) implemented to use the model to improve the decision of resource allocation. This work is one of the pillars of our work, and will be extensively explained in further chapters.

1.1 MOTIVATION

In the literature, performance estimation tools for computing systems are mainly of two category: high-level models and precise cycle accurate simulators (KUNKEL et al., 2000). The second category give almost exact results, one example of it was implemented by Bergamaschi et. al. (BERGAMASCHI et al., 2007), resulting in a tool named SLATE (System-Level Analysis Tool for Early Exploration). This tool presented nearly exact results, but it is very time consuming, being helpful to orient computing systems design for example, since it can predict the actual task performance and point out bottlenecks. This result in a tool not suited to quick decisions that a live running system sometimes have to do, for example scheduling of tasks or thread switching.

On the other hand, the high-level models do not touch the behavior of the processors. These models are based on empirical observations of the overall system performance, or are based on abstract models (that can be split in several loosely coupled models). One example of this approach was proposed by Torrelas et. al. (TORRELLAS et al., 1990), where a queueing network model of the memory architecture of multi-CPU systems showed good results for some class of tasks. One problem with this approach is that some parameters in the model are not directly measured, but rather are estimated. Other problem is that the actual CPU role is not take into acount, making the model not suitable to simulating programs purposes. Refinements on the model were done resulting in SPLASH (Stanford Parallel Applications for Shared-memory) (SINGH et al., 1992) that is a set of parallel applications for use in the design and evaluation of shared-memory multiprocessing systems.

In our point of view, a new alternative to approach the CPU performance model is the work by Kandalintsev named *"A behavioral first order CPU performance model for clouds management"* (LO CIGNO; KANDALINTSEV, 2012). This simple model proposes a CPU centric model, based only on measurable parameters, capable to produce better approximations of the real systems performance than the actual linear CPU scaling performance paradigm usually present in the literature (JIANG et al., 2012), (SHRIVASTAVA et al., 2011), (TANG et al., 2007) and (FELDMAN et al., 2009). The model avoids to touch CPU internals, what makes it computationally very fast and suitable to be implemented in simulators, since it is possible to model the CPU performance individually.

The idea of being possible to simulate the CPU performance when running different tasks is one key driver in the present work. The idea is that if we modify an existing simulator to implement the Kandalintsev's model, it could enhance the results giving a better simulator platform to researchers and developers to implement their own algorithms (like schedulers, load balancers, and so on). Reviewing the literature, we choose CloudSim simulator (CALHEIROS et al., 2011), because it is flexible enough to be extended, it is free software, well documented and most of all (besides being created to Cloud Simulations purpose) it could be adapted to simulate what we were interested, that is the CPU scaling performance in multi-core computational systems.

1.2 OBJECTIVES

1.2.1 GENERAL OBJECTIVES

The main objective of this work is to extend the CloudSim Simulator, implementing into it the BFO (Behavorial First Order) performance model of Kandalintsev and Lo Cigno. This objective is divided in two sub-objectives, the first is to improve the accuracy of the results presented by the original CloudSim in terms of the actual performance of real multi-core systems. The second objective corresponds to propose an extended model to explicitly consider te number of cores that work in parallel. The original BFO model presents an elevated level of error , specially when the number of parallel cores are higher than 8 cores. Our extended model can improve the simulations results from CloudSim.

CloudSim is a good choice because it follows a common paradigm, that the CPU performance scaling is linear, that is reasonable considering that it is very common that the systems, specially in Clouds that are the focus of CloudSim, are consuming little computation power compared with what the system has available. This will be used in favor of this research, because we have granted a simulator that has no performance correction implemented, and we guarantee that our implementation is unique for this purpose.

The BFO model proved to be a good choice, because it can give us the baseline to compare the real performance of multi-core systems against the simulator (calibrating it), and then use it to validate the implemented model when the number of cores raise or the size of the problem increases or decreases. It is expected that the result of these cycle (implementation of the model, training of the simulator, generalization of use cases in the simulator, comparison with real systems measurements) will lead us to an implementation adherent with real world scenarios, as well as with a method to serve anyone interested in implement its own research in this area.

1.2.2 SPECIFIC OBJECTIVES

The specific objectives of this work are:

1. To validate the BFO model by comparing it against results from an experimental system. This is important to gain experience on the performance counters available in the processors, as well as the correct use of the tool to generate the load for the CPUs: the

stressors *stress-ng* (UBUNTU-WIKI, 2008), and the tool to read the counters *perf* (WIKI, 2015).

2. To implement the BFO model in the CloudSim simulator. This is made via extension of some classes and creation of brand new classes inside the CloudSim package.
3. To validate the extended CloudSim against the real world systems data collected.
4. To use the extended CloudSim to predict the behavior of different loads patterns.
5. To propose and test an extended model that considers explicitly the number of cores that work in parallel.

1.3 STRUCTURE OF THE DISSERTATION

This dissertation has six chapters (Introduction, Parallel Processing Performance Models, Performance Modeling, Extensions on CloudSim to Implement the BFO Model, Framework for Multi-Core Systems Evaluation, Conclusions and Future Research).

In the chapter *Parallel Processing Performance Models*, we describe in detail some important models to characterize the CPU escalation problem, that is the Amdahl's law (or Fixed Size law), the Gustafson's law (or Fixed Time law) and the Sun and Ni's law (or Memory Bound law). These are statistical models constructed to speculate about the scaling capabilities of a parallel computational system, and the multi-core processors are such a system.

In the chapter *Performance Modeling* starts with a bibliographic review where the cycle accurate and high-level models are explained and one of a kind in described, the RPM (a high-level model) and the SLATE (a cycle-accurate model). In the sequence, the model created by Kandalintsev and Lo Cigno is explained in details, as well as it is described how to obtain the coupling factor β using off line procedures.

In the chapter *Extensions on CloudSim to Implement the BFO Model* we see the actual implementation of the BFO model in CloudSim. It is shown the UML diagrams with the detailed implementation of the model. The first part shows the CloudSim core simulation engine, and the second part is dedicated to the implementation.

The chapter *Framework for Multi-Core Systems Evaluation* is dedicated to explain the methodology followed to train and to test the model using measurements done in real world systems. It is done as a comprehensive guide on how the tests and corrections were made to train the simulator and the use of it to extend the results, comparing it with the real world computational systems. In addition, at the end we propose an extension of BFO model based in a logarithmic rule to correct the coupling factor as a function of the number of parallel cores active. This reduces the RMSE error that grows with the number of cores.

In the chapter *Conclusions and Future Research* are compiled all the findings, as well as, the observations and the insights the research leaded us.

2 PARALLEL PROCESSING PERFORMANCE MODELS

2.1 PERFORMANCE IN MULTI-CORE PARALLEL SYSTEMS REVIEW

By empirical measurements we know that when working with multi-core CPUs the overall performance (speed up) does not grow linearly with respect to the number of available cores. This is related to use of shared resources during runtime, for example, internal caches, bus, interfaces, I/O, RAM, and network. Because of it, the cores cannot run independently making the combined performance (speed up) be less than the number of number of cores working in parallel.

This problem was first addressed by Amdahl (AMDAHL, 1967) and leaded to a discouragement of manufacturers in the earlier times of the computer industry. Some decades after that, Gustafson (GUSTAFSON, 1988) revisited the problem because the micro-architecture of modern computation had overcome some of the architectural problems saw by Amdahl, showing that for multi-CPU scenario the performance scaled less restrictively than Amdahl's assumptions, arguing that software designers adapt their algorithms to scale down the serial part in a way that the execution time is fixed. After some years Sun and Ni (SUN; NI, 1990) showed how memory plays an important role on the performance, assuming that the execution of an algorithm scales in the memory (specially caches), the model is known as memory bound. This model is promising and has influenced the foundations of modern multi-core architectures in a sense to improve memory access avoiding what is known as memory wall problem. As one of the goals of the actual work is to predict the CPU scaling based on a behavioral model where the inputs are collected by data collected in an experimental system, a complete understanding of those three laws is of fundamental importance, since it gives us the ground to discuss the results and what can be improved.

2.2 AMDAHL'S LAW, DEFINITIONS AND CONSEQUENCES

The first work that called attention to the scalability problem in parallel computer systems was done by Eugene Amdahl (AMDAHL, 1967), and consists of an observation that the execution time is formed by two parts, the one that benefits with the parallel processing, and the other that is irreducible and does not benefit with the parallel processing. He divided the processing timespan in two parts, the *parallel* that is related to the amount of time that the CPUs run independently, with no conflicts with the other CPUs, and can be expressed as the ratio p with $p \in [0,1]$. The second part is related to amount of time that cannot be reduced by the parallelism, due to the conflicts in the shared resources, this is called *serial* execution time and can be expressed as the ratio s with $s \in [0,1]$ and $s = 1 - p$. Equation 1 shows the mathematical formulation of the Amdahl's law, where the *speedup* $S(n)$ is defined as how much faster a parallel arrangement will get if compared with the same problem is solved with a single CPU. This law is a pure statistical description of the problem when all the multiple CPUs are supposed to be running at maximum capacity (i.e. 100%).

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}} \tag{1}$$

When Amdahl put his observations the Main Frame vendors were discussing the possibility of work with multiple CPUs in parallel to scale up the processing power for customers. The result stopped the parallel CPUs approach. At that time the parallel ratio was estimated to be around 70%, leading to a maximum speed up of 3.33. With the single CPU approach wining, all the engineering efforts were put to develop CPUs increasingly powerful, pushing the engineers and scientists to keep in this direction for decades. What kept the industry drive was the *Moore's law*, that is an observation made by Gordon Moore, where the number of transistors roughly doubles in the integrated circuits every two years (MOORE, 1965). To unveil the potential of this claim one have to combine *Moore's law* with *Polllack's Rule* (POLLACK, 1999) that states that for a single core microprocessor the performance increases roughly proportional to the square root of the increase in its complexity. What leads a performance increase of about 40%, each two years, due to the doubling in number of logical ports.

However, recently Moore's law has been losing its momentum, generating a profound impact in the electronic and software industry. Conte (CONTE et al., 2015) has affirmed that: "Year-over-year exponential computer performance scaling has ended. Complicating this is

the coming disruption of the “technology escalator” underlying the industry: Moore’s law. To fundamentally rethink and restart the performance-scaling trend requires bold new ways to compute and a commitment from all stakeholders”.

New alternatives come in place to keep the law working, even *“artificially”*, and the parallel processing via multiple CPUs working together came back again as a solution. But as we know the performance will not scale linearly with the number of CPU cores. Therefore, the characterization of the real performance of CPU cores running in parallel has become very important to keep the exponential growth escalator of Moore’s law still working.

2.2.1 AMDAHL’S LAW AND SPEEDUP IN MULTI-CORE SYSTEMS

Amdahl’s law presents an asymptotic behavior as shown in Figure 1 (speed-up for parallel ratios of $p = 0.7$, $p = 0.9$, $p = 0.95$ and $p = 0.995$). Equation 2 shows that the maximum speed-up depends only on the parallel ratio.

$$S(n)_{max} = \lim_{n\to\infty} S(n) = \lim_{n\to\infty} \frac{1}{(1-p)+\frac{p}{n}} = \frac{1}{(1-p)} \tag{2}$$

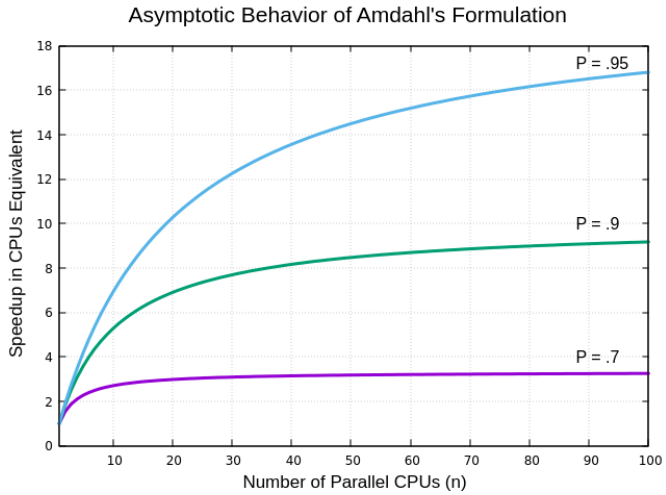


Figure 1: *Speedup* dependency on number of CPUs (n) and Parallel Ratio (p).
Source: (Based on Equation 1)

Synthesizing this behavior, no matter how many parallel cores we have in a system, the overall speedup is limited to a maximum value, that is dependent on how efficiently parallelized the system can be.

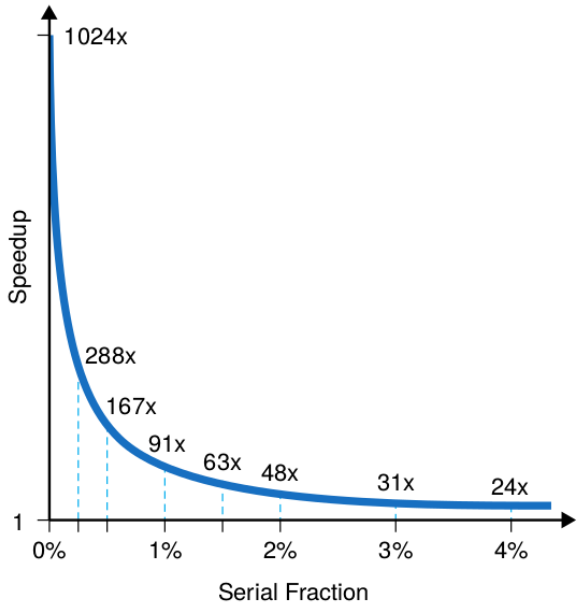


Figure 2: Asymptotic maximum *speedup* function of the serial ratio *s*.
Source: ((GUSTAFSON, 1988))

Another way to see it is shown in Figure 2, where the maximum speedup is plotted as a function of the serial ratio s. Basically it plots the result of Equation 2 for each $s = 1 - p$. Note that when s is small, the speedup grows quickly, what is already expected since with a smaller s the more efficient is the parallel part of the system.

2.2.2 AMDAHL'S LAW COROLLARY - THE FIXED SIZE LAW APPLIED TO MULTI-CORE

To understand the consequences and the opportunities derived by Amdahl's Law when applied to multi-core architecture Hill and Marty (HILL; MARTY, 2008) have derived a corollary, where the micro-architecture of multi-core systems is further explored to refine the results of Amdahl's law in such environment. The speed-up reflects the timespan reduction when the task is running in a multi-core system compared to the same task running on a single core. For now we will explore this assumption to formulate the speedup for multi-core systems, but the reinterpretation of this assumption is the basis for the work of Gustafson (GUSTAFSON, 1988), that better explain the behavior of highly parallelized systems.

The formulation of the corollary starts with the assumption that a multi-core chip is formed by n cores, and each core can individually deliver a processing power of one BCE (Base Core Equivalent), meaning that all the cores are identical. At this point, the speedup is measured in BCEs, as one single core has one BCE, the speedup of a multi-core is the total

amount of BCEs for a particular micro-architecture and this is limited to n BCEs (when the $s = 0$). The other assumption is that the architects can come up with more efficient systems that can treat the serial part with a performance that scales in the order of $perform(r)$ (in other words, the sequential part can be seen as a single processor with r BCE processing power). In addition, if r BCEs are spent to the serial part treatment, the parallel part can scale up to $m = n/r$ BCEs. According to Bokar (BORKAR, 2007) the serial performance can be considered $perform(r) = \sqrt{r}$, that is an assumption coherent to Pollack's Rule (POLLACK, 1999).

To continue the analysis, we have the definition of fixed size speed-up shown in Equation 3, meaning that the performance of the parallelized system (enhanced) is compared with the single core system (original). To calculate the speedup, the performances are calculated as the reciprocal of the execution times.

$$Speedup_{FS} = \frac{Enhanced\ Performance}{Original\ Performance} = \frac{T_{Original}}{T_{Enhanced}} \tag{3}$$

Considering that the task has a fixed size w, we have in consequence the execution times presented in Equations 4a and 4b:

$$T_{Original} = \frac{w}{perform(1)} \tag{4a}$$

$$T_{Enhanced} = Serial\ time + Parallel\ time = \frac{(1-p)w}{perform(r)} + \frac{Pw}{\frac{n}{r}.perform(r)} \tag{4b}$$

Combining Equation 3 with Equations 4a and 4b, we end up with a $Speedup_{FS}$ shown in Equation 5:

$$Speedup_{FS} = \frac{\frac{w}{perform(1)}}{\frac{(1-p)w}{perform(r)} + \frac{Pwr}{n.perform(r)}} = \frac{1}{\frac{1-p}{perform(r)} + \frac{p.r}{perform(r).n}} \tag{5}$$

This formulation seems unrelated to Amdahl's law, but if we consider that the enhanced system can deliver a constant performance $perform(r) = c$ and that $m = n/r$, the Equation 5 becomes:

$$Speedup_{FS} = \frac{c}{(1-p) + \frac{p}{m}}\ , \tag{6}$$

that is exactly the original Amdahl's law with the serial scaling of r BCEs and the parallel processing scaling of m BCEs.

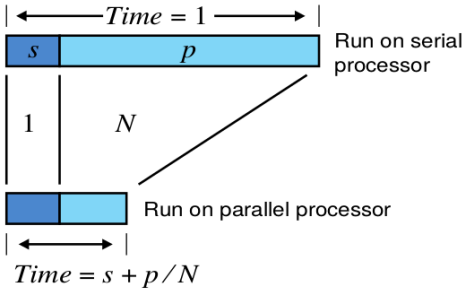
That is why Equation 5 is a corollary of the original law. This result is important to send a message to micro-architects to develop multi-core systems that can handle the serial part more efficiently as well as support non blocking parallelization. This would lead to better performance. The great contribution of Hill and Marty work is the baseline to think in a more restrictive environment that is the micro-chips, where the degrees of freedom are diminished, due to topological and physical reasons.

2.3 TIME FIXED LAW, THE GUSTAFSON'S LAW

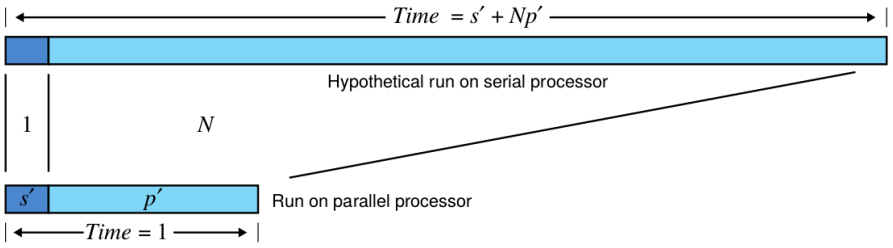
The problem with Amdahl's law and its corollary is that it is too restrictive. Empirically we know that the performance scaling is not linear, but it is not as bad as the predictions we have so far. This was the point of start for Gustafson's work (GUSTAFSON, 1988) that observed it and conjectured that the fixed size approach is not dominant in modern architecture multi-CPUs. According to him literally: *"We now have timing results for a 1024-processor system that demonstrate that the assumptions underlying Amdahl's 1967 argument are inappropriate for the current approach to massive ensemble parallelism."* and *"Yet for three very practical applications (s = 0.4 – 0.8 percent) used at Sandia, we have achieved the speedup factors on a 1024-processor hypercube which we believe are unprecedented: 1021 for beam stress analysis using conjugate gradients, 1020 for baffled surface wave simulation using explicit finite differences, and 1016 for unstable fluid flow using flux-corrected transport. How can this be, when Amdahl's argument would predict otherwise?"*

Of course, the kind of interconnection architecture cited ("hyper-cube") is not easy, if not impossible, to reproduce in microprocessors that are essentially planar. However, he started an argumentation that puts more light on the study of the real behavior of the speedup that is more adherent to the actual state of art in microprocessors. It was proposed that the tacit assumption that the parallel workload p is fixed independently of the number of available cores N is not always true, but actually the problem size scales with the number of processors. It is because when a more powerful processor is used, the problem expands to the user to make use of the increased facilities (for example, grid resolution, number of time-steps, difference operator complexity, and other parameters) making the time to complete the task approximately constant. This assumption is know as Fixed Time and relaxed the Fixed Size assumed by Amdahl.

Figure 3(a) and Figure 3(b) shows both models, Fixed-Size and Fixed-Time, with the resulting speedup. To derive the Fixed-Time speedup imagine that the workload w running on



(a) Fixed-Size Model: $Speedup_{FS} = 1/(s+p/N)$



(b) Fixed-Time Model: $Speedup_{FT} = s+Np$

Figure 3: Scalability of Fixed-Size (Amdahl's law) and Fixed-Time Models (Gustafson's law). Source: (GUSTAFSON, 1988)

a single CPU scales up to w' on a multi-core processor, with the assumption that the time in each core is fixed (normalized to 1), we have thus $w' = (1-p)w + pNw$. Equation 7 shows the $Speedup_{FT}$, also known as Gustafson's Law:

$$Speedup_{FT} = \frac{Sequential\ Time\ of\ Solving\ w'}{Parallel\ Time\ of\ Solving\ w'} = \frac{Sequential\ Time\ of\ Solving\ w'}{Sequential\ Time\ of\ Solving\ w} = \frac{w'}{w}$$
$$= \frac{(1-p)w + pNw}{w} = (1-p) + Np = s + Np \tag{7}$$

Equation 7 suggests that there is a linear scaling of speedup that we know is not true, but shows better results when considering experimental data. So something is missing on this law, that will be explained in the next section, with a refination of this model, known as Sun and Ni's law (or Memory Bounded Model). From now, it is important to get a better understanding of the techniques to calculate the scaling, so we will have the arsenal to understand next section.

For example, if $n = r$ in Equation 5 the speedup is equal to $perform(r)$, this is used as

the initial condition for the analysis that follows. Now if $n = mr$ is the scaled number of cores to be used in the fixed time model approach (that assumes the scaling is only in the parallel portion) thus $w' = mw$, then we have the total workload:

$$\frac{(1-p)w}{perform(r)} + \frac{pw}{perform(r)} = \frac{(1-p)w}{perform(r)} + \frac{pw'}{perform(r)m} \tag{8}$$

Comparing Equation 8 with the scaling of serial part, that is the initial situation where $n = r$, we have:

$$Speedup = \frac{Sequential\ Time\ of\ Solving\ w'}{Sequential\ Time\ of\ Solving\ w} = \frac{\frac{(1-p)w}{perform(r)} + \frac{pw'}{perform(r)m}}{\frac{w}{perform(r)}} = (1-p) + mp \tag{9}$$

To analyse what Equation 9, if we consider a situation where 1024 cores are working in parallel, and considering $p = 0.99$ in fixed time model, we will have a speedup of about 10. On the fixed-time model it jump to about 1013. What is behind the assumption of a fixed time model is that there are always room for more work to process in the parallel processing part. To illustrate it, let's see what happens when we escalate the parallel part during the scaling up the number of cores:

$$Total\ Work = Serial\ Work + Parallel\ Work = (1-p)w + pw' = [1 + (m-1)p]w \tag{10}$$

Once the parallel work is escalated, compared with the total work calculated by Equation 10, the new parallel ratio p' is shown on Equation 11:

$$p' = \frac{mfw}{[1 + (m-1)p]w} = \frac{p}{\frac{1}{m} + \frac{m-1}{m}p} \tag{11}$$

If $m \to \infty$ in Equation 11, then $p' \to 1$, showing that the serial part is less and less important in the scaling process.

2.4 MEMORY BOUND MODEL, THE SUN AND NI'S LAW

The Gustafson's assumption that the parallel part scales up to meet a fixed time bound is not always true, and Sun and Ni (SUN; NI, 1990) observed that the parallel scaling problem was related to the memory scaling capabilities of the systems. This is a natural approach, if we take the assumption that it is impossible to put infinite resources in the chip, so there is a bounding in the scaling capabilities of the computational system imposed usually by memory

limitations.

Sun and Ni proposed the Memory Bound Model, for that they defined the $Speedup_{MB}$ as shown in Equation 12, where w^* is the scaled workload under a memory space constraint, in other words the *memory is bounded*.

$$Speedup_{MB} = \frac{Sequential\ Time\ of\ Solving\ w^*}{Parallel\ Time\ of\ Solving\ w^*} \tag{12}$$

If we assume that a single core is a pair *processor-memory*, once we scale up the number of cores, the memory will also scales up. Letting $y = g(x)$ represents the parallel part p, the workload increases when the memory capacity increases for a factor of m. Implicating that $w = g(M)$ and $w^* = g(m.M)$, with M being the amount of memory for one processor-memory pair. We can say that $w^* = g(m.g^{-1}(w))$. Thereby Equation 13 shows the generic $speedup_{MB}$ for any $g(x)$.

$$Speedup_{MB} = \frac{(1-p)w + p.g(m.g^{-1}(w))}{(1-p)w + \frac{p.g(m.g^{-1}(w))}{m}} \tag{13}$$

It is important to say that $g(x)$ is a function that depends on the application running in the computational system, because it reflects the scaling that the memory experiments when running the application, that is essentially connected with the essence of the algorithm being used by the application. In other words, scaling is function of the kind of application running. It is confirmed by the empirical data collected for the present work.

The generic formulation of Memory Bound Model was further developed by Sun and Ni considering that there are many algorithms with polynomial complexity, and usually this complexity is fully characterized using only the highest term of the polynomial. This way, considering $g(x) = a.x^b$, with $a, b \in \mathbb{Q}$. Thus we can say that $g(m.x) = a(m.x)^b = m^b.a.x^b = m^b.g(x)$, now calling $m^b = \bar{g}(m)$ as a power function with degree of 1. Equation 13 becomes Equation 14 that is also know as *Sun and Ni's Law*.

$$Speedup_{MB} = \frac{(1-p)w + p.\bar{g}(m)w}{(1-p)w + \frac{p.\bar{g}(m)w)}{m}} = \frac{(1-p) + p.\bar{g}(m)}{(1-p) + \frac{p.\bar{g}(m)}{m}} \tag{14}$$

To gain insight on what Equation 14 represents and how to use it, consider the situation where the application to be studied is an algorithm for matrix multiplication. The complexity of this algorithm when solved by brute force is polinomial with degree 3, actualy $y = 2N^3$ with

N meaning the dimension of the two NxN matrices being multiplied. We can say that $x = 2N^3$ what leads to $g(N) = 2(\sqrt{\frac{x}{3}})$ and $\bar{g}(x) = x^{\frac{1}{2}}$. Puting these results on Equation 14, we have the Equation 15:

$$Speedup_{MB} = \frac{(1-p) + p\bar{g}(m)}{(1-p) + \frac{p\bar{g}(m)}{m}} = \frac{(1-p) + p.m^{\frac{3}{2}}}{(1-p) + p.m^{\frac{1}{2}}} \tag{15}$$

This clearly is not the linear scaling predicted by Fixed Time Model. If we consider the work-flow scaling in Memory Bound as w^* and in Fixed Time as w', if each element of the matrix that is stored in memory is used at least one time, then $w^* \geq w'$ what leads to $Speedup_{MB} \geq Speedup_{FT}$. In fact Gustafson's law is a particular case when $\bar{g}(m) = m$.

Initially this is good news for the multi-core solution, but there is a drawback, as said before, there is a physical limitation with the amount of memory available and the access time to treat this scaling problem, it is know as *memory wall problem* (WULF; MCKEE, 1995). In simplistic words, one can say that there is no room in local memory (specially caches) to guarantee the locality (cache hits) during all the execution time. This means that, delays to retrieve all the needed data plus instructions plays important role when the size of the problem grows. To take it in account in the next session the memory wall is considered in the model.

2.5 MEMORY WALL PROBLEM

The access time to memory has big impact on performance. Over the time, this impact has became much worse, because when we compare the access time to memory with respect with the clock cycle period, it has increased several orders of magnitude. It has certainly decreased in terms absolute values, but relative to the clock cycle (that dictates the instruction execution pace) it increased significantly. For example, in 1980 the memory latency was about 0.3 cycles of clock, in 2005 this value has jumped to 220 cycles of clock.

To mitigate this effect and improve the performance of the CPUs, memory caches were aggregated into the CPU chips. In multi-core chips, caches are distributed alongside the cores. These cache memories are much faster than normal memories, but they are much smaller because of manufacturing costs and chip space. This approach works well because it relies on the locality of the software, meaning that must of the time processors are executing repeatedly a small segment of instructions and data. For example, most of the time the algorithms are executing *"for loops"*.

But this architecture has a drawback, if the information needed is not in the cache, the processor first look it in the cache, receiving a cache miss information, and only after screening all cache levels (L_1, L_2 and L_3) it is allowed to seek the information in the external memory. It makes the external memory access even slower. This architecture is usually more efficient than not using caches. However, in highly parallel systems, if we don't have enough cache to respond to the scaling of the problem size, it can create a bottleneck to the CPU scaling for the parallel tasks.

Figure 4 shows the memory organization in multi-core systems, caches L_1 are the closer to the core, L_2 caches can be shared among pair of cores. L_3 cache can exist or not, if not existing it is the external memory (usually DRAM).

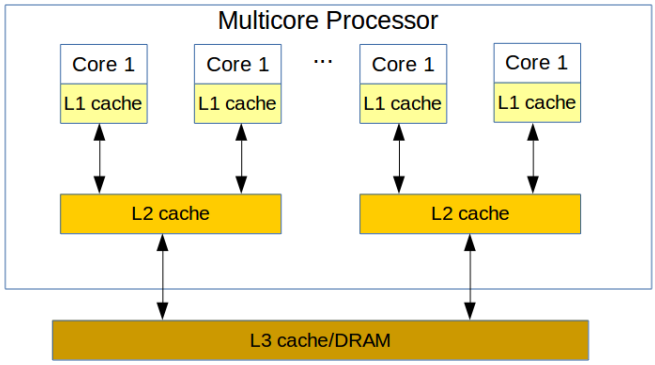


Figure 4: Typical Memory Organization in Multi-core Systems.

The diminishing CPU scaling due to lack of cache in multi-cores is known as the *memory wall problem*, first introduced by Wulf (WULF; MCKEE, 1995). To have an idea of how this impacts the system scaling, an Intel i7-6500U implements L_3 cache size of 4096kB, L_2 caches with size of 256kB and L_1 caches with size of 32kB. It has basically the same core architecture of a Xeon E7-8890 v4, nevertheless Xeon will have 60MB for L_3 cache, 4MB for L_2 and L_1 caches. But Xeon scales much better then i7, being a processor of choice for Servers and specialized systems, with i7 being more suitable to personal computer systems.

The *memory wall* scaling can be modeled using the same approach used to model Fixed Time and Memory Bound speedups. In this case we assume that the total work for a task is composed of two parts, the data processing work w_p and the data communication (access) work w_c, and $w = w_p + w_c$.

If assumed that w_c is independent of the workload and independent of the total workload (what is reasonable since the memory access time and the lines of code are The same independently to the number of cores for a particular situation), and w_c is function of r and supposing the scaling of the number of cores from r to mr, as per Memory Bound model we will have :

$$Speedup_{MB} = \frac{\frac{w_c}{perform(r)} + \frac{pw_p^*}{perform(r)}}{\frac{w_c}{perform(r)} + \frac{pw_p^*}{m.perform(r)}} = \frac{w_c + pw_p^*}{w_c + \frac{pw_p^*}{m}} = \frac{w_c + p.g(m.g^{-1}(w_p))}{w_c + \frac{p.g(m.g^{-1}(w_p))}{m}} \tag{16}$$

In Equation 16, if $g(x)$ is rational, we have the simplification bellow:

$$Speedup_{MB} = \frac{w_c + p.\bar{g}(m).w_p}{w_c + \frac{p.\bar{g}(m).w_p}{m}} \tag{17}$$

Equation 17 can be compared with Equation 14 to have an idea of the memory wall impact over the speedup due to the effect of memory scaling and latency. What we see is that w_c dominates the serial workout, meaning that the scaling of w_p can reach a saturation. This leads to an obvious conclusion, if is to put effort to keep improving the hardware efficiency, research in memory improvements should be a focal point.

First of all, it is important to say that all these models are essentially right if they have their assumptions met. The models were constructed to speculate and further advance the knowledge on what is important and what can be neglected for parallel computing, regarding the CPU scaling. In addition, these models give a ground to think where to concentrate efforts in architecture research and development to extract the best performance. However, these models are not enough to calculate a real world scaling problem, because they were created to give general answers and to support benchmarking. In the next chapter we will address a model form Kandalintsev and Lo Cigno (LO CIGNO; KANDALINTSEV, 2012), that was developed to address the problem of predicting the real scaling based on measured coupling factors.

3 PERFORMANCE MODELING

3.1 MODELING FOR PERFORMANCE PREDICTION

In the previous chapter we saw statistical laws that help to understand the performance, in terms of CPU scaling, considering the time-span divided in two parts, the serial and the parallel ratio. This was useful to understand the play role of the macro components (architectural and software design) and the impacts on the overall system performance. However, in practice, the execution ratios are not known beforehand, and they are not easily measured, since they are dependent on the nature of hardware design and its interaction with the software it is running. This opens the necessity for models and tools capable to predict the performance based on the behavior of some components. There are several of such tools and models, according to Kunkel et. al. (KUNKEL et al., 2000) they usually fit in two types, *cycle-accurate models* and *high-level models*. Unfortunately, for the research goals none of these approaches are good fits, because our idea is to have a performance model capable to assist on real time decisions made by the Operating System or a Resource Management software.

Bellow we describe one *cycle-accurate model* and one *high-level model*, it will give us a good ground to understand why the BFO model proposed by Kandalintsev and Lo Cigno is better suited for our purposes.

3.1.1 RPM - HIGH-LEVEL MODEL

RPM (Rapid Prototyping Engine) (BARROSO et al., 1995) is a *high-level model* tool designed to multi-core prototyping. It is based on a hardware emulation of the processor. It is build using FPGA (Field Programmable Gate Array) to build the cache, memory, coherence and communication controllers, and some off-the-shelf components including processors, SRAMs, DRAMs, FIFOs, bus interface and drivers, and backplane.

The idea is to put the original processor served with the FPGA services instead of the internal cache or the coherence and communication controllers. This way it is possible to register the statistics of events not accessible in multi-core processors, that are crucial to model the performance behavior in a very accurate way. This approach is very powerful to benchmark one architecture running a real application. However, RPM emulates the cache operations 80 times slower than the original computational system.

Figures 5 and 6 shows the First-Level Cache (FLC) and Second-Level Cache (SLC) implementation for RPM. In Figure 5, the FLC controller is implemented in FPGA1 and FPGA2 (in Memory Controller 1 (MC1)), they connect to the Address and Data Bus of the processor, in this case a SPARC L64831. When the processor read the L1 cache, the cache controller answer in Data Bus in case of a cache hit, otherwise it seek for the SLC using the cache controller Address Bus. Figure 6 details the SLC controller architecture. Again, if the there is a cache hit, Memory Controller 2 (MC2) deliver the data segment to MC1 (it is based on a send/receive protocol implemented in hardware). In case of cache miss, the SLC controller will seek for a new segment in the external memory. This is a known process, but the details of implementation can be studied to improve the performance.

Moreover, the FLC and SLC controllers work in a different clock, usually 8 times faster than the processor clock (*pclock*). It is useful to get the details of the process and study the intermediate states. However, the FPGA maximum clock is usually less than the processor maximum clock, usually in a factor of 10, and added to it *pclock* is set 8 times slower than the FPGA clock, it means that this setup runs 80 times slower than the processor in a production environment. It makes this approach, very usefull for prototyping and benchmarking, but not usefull for real time performance measurements.

The same approach, high level emulation, is implemented by several authors/tools. For example Eyerman et. al. (EYERMAN; EECKHOUT, 2009), developed a methodology to study SMT (Simultaneous Multi-Thread) processors. It uses the same approach of RPM in a sense that the cache study is central in the analysis. However, modern multi-core processors have internal counters that can be read in real time, making it possible to infer the relevant statistics, instead of directly measuring it as RPM does.

Anyways, besides very accurate, these models are time consuming and not suitable for real time decisions. They best fit for benchmark, as stated in the introductory paragraph of the

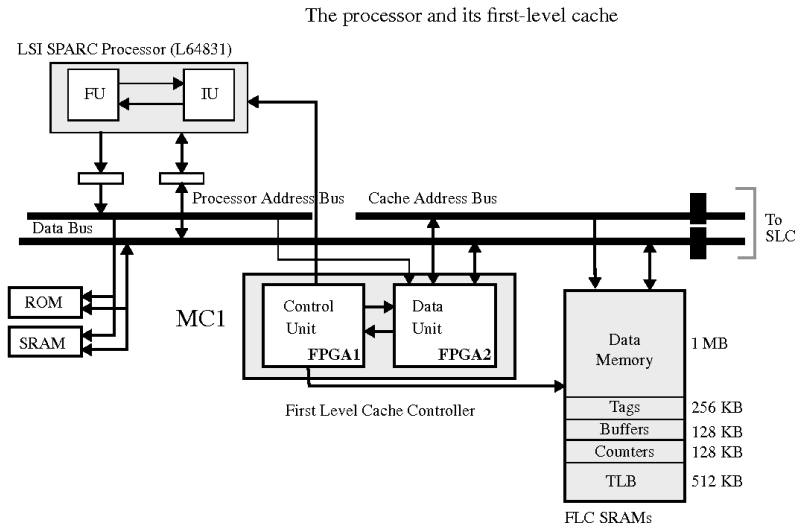


Figure 5: RPM First-Level Cache (FLC) implementation diagram; it shows the L1 cache implemented in FPGA and the Bus coherence to access L2 cache (refer to Figure 6 for details).
Source: (BARROSO et al., 1995)

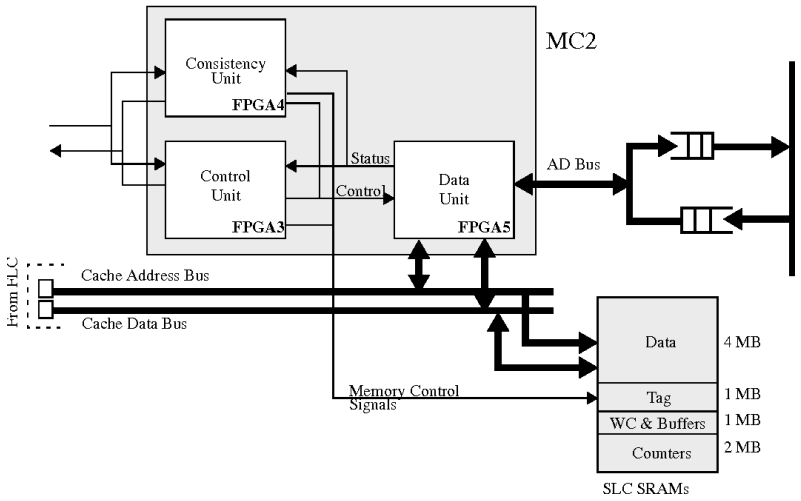


Figure 6: RPM Second-Level Cache (SLC) implementation diagram; it shows the L2 cache implemented in FPGA and the Bus controller to access external Memory (AD Bus).
Source: (BARROSO et al., 1995)

section.

3.1.2 SLATE - CYCLE-ACCURATE MODEL

SLATE (System-Level Analysis Tool for Early Exploration) (BERGAMASCHI et al., 2007) is a tool based on a methodology that models the CPU core internals, based on execution and wait times, in a queue driven analysis. It is based on SystemC performance modeling, that consists on the description of the model to be simulated in terms of components (caches, queues, cores and interconnection between them). In each component it is associated the times.

For example, cores have execution time and caches have wait times (associated to the time to make the data available). Figure 7 and Figure 8, shows a complete Power4 CPU modeling. The Power 4 is a quad-core, single thread, single instruction CPU, with L2 and L1 caches implemented. In Figure 7 it is modeled a single core and its components (decode, dispatch, issue queues, execution units, and so on). All are modeled in high-level in terms of delays and execution times, and the engine calculates cycle by cycle the state of the system (that is why it is cycle accurate model). The modeled core is then used as the building block of the model in Figure 8, where are modeled the interconnections and the controls of the multi-core environment.

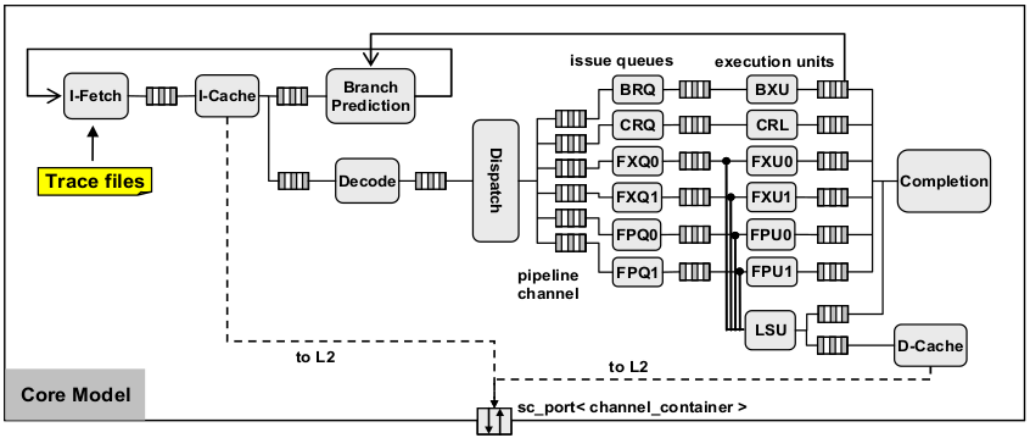


Figure 7: Internal Organization of Core Model - Power4 single core modeled.
Source: (BERGAMASCHI et al., 2007)

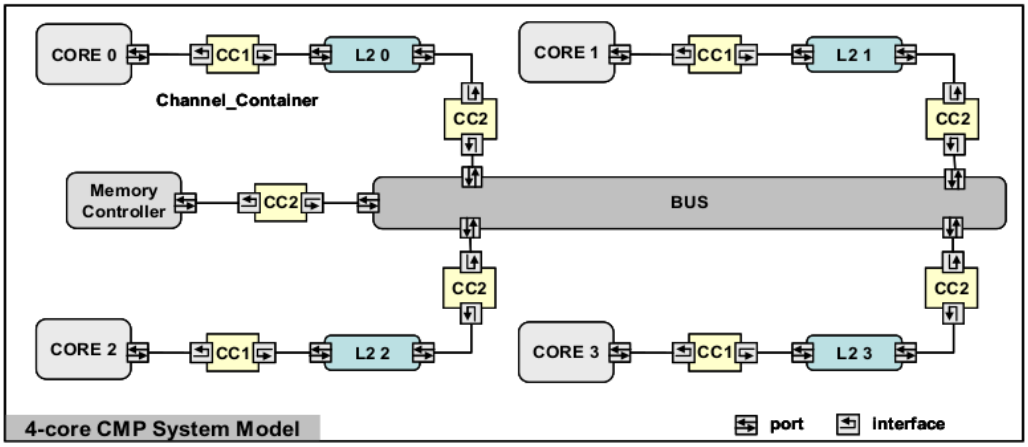


Figure 8: Internal Organization of 4-Core System Model - Power4 CPU modeled.
Source: (BERGAMASCHI et al., 2007)

Cycle-accurate models are useful to prototype the CPU design, to predict the behavior of the architecture under test with predefined applications modeled in terms of load and size. It

is not suitable to our purposes, since our idea is to simulate the multi-core performance scaling of commercial processors and not prototype a new processor.

3.1.3 THE BEHAVIORAL FIRST ORDER (BFO) CPU PERFORMANCE MODEL

In Kandalintsev's PhD Thesis *"Application Interference in Multi-Core Architectures: Analysis and Effects"* (KANDALINTSEV, 2016), Chapter 3 is dedicated to explain the BFO model conceptually and mathematically. He observes that it is well known that the interference between cores plays an important role in multi-core processors, for example, Burger et. al. (BURGER et al., 1996) explains how the velocity gap between processor speed and memory access has increased and the consequences in performance of processing, similar work is developed by Carvalho et. al. (CARVALHO, 2002) pointing that performance of modern CPUs are dependent on the performance of internal cache that are shared by the cores. Xu et. al. (XU et al., 2012) studied how to minimize the impact of the interference among the cores studying algorithms to provide fairness in the use of caches.

Based on a inter-core interference interpretation, Kandalintsev proposed a model that studies the decrease in performance of a CPU core, due to concurrency with others CPU cores for common shared resources. It models the observable behavior, the increase in execution time when the CPU core is competing with others for shared resources, comparing with the situation where the CPU core presents no concurrency. The model defines the extra time to complete a job when concurrency is present as *overhead* (OH).

The model proposes that the overhead will be measured when the CPU cores under test are fully occupied (100% load), but admits that the actual OH is a function of the actual load in the CPU cores. In other words, if the CPU cores are using less than 100% of capacity, they will have lesser probability to find some resource already occupied. This assumption is compatible with the Amdahl's law corollary, where the speed-up of multi-core systems are better than the Amdahl's law predicts (in terms of BCEs) if the actual utilization is lower than what is available in the processor. In fact it is expected that the serial ratio be lowered by the performance function $perf(n)$ and it is accepted to be $perf(n) = \sqrt{n}$ by experience, being n the actual amount of BCEs demanded.

Starting now with the formal definitions of the model we have:

1. *Assumptions:* The overhead is present when there are tasks running in parallel, otherwise there is no overhead. The model considers steady state performance (meaning tasks that run for long time), examples are video encoding, scientific simulations and data streaming. It is assumed that the dominant interaction between tasks is pairwise, i.e., the total overhead is the sum of the overhead running in pairs. The expectation here is to have the overhead overestimated, but still a good approximation of the real system.

2. *Definitions and Notation*: The task load, $L_i = L(T_i)$ with $L_i \in [0,1]$, is defined as the time required by task i in a particular CPU core, assuming that the CPU core can be running several tasks in time shared way. The total load per-core is the sum of the task loads and sum up to a maximum of 1. Figure 9 presents a graphical view of load task and how to calculate it. N is defined as the number of CPU cores. L_{sys} is the *normalized system load*

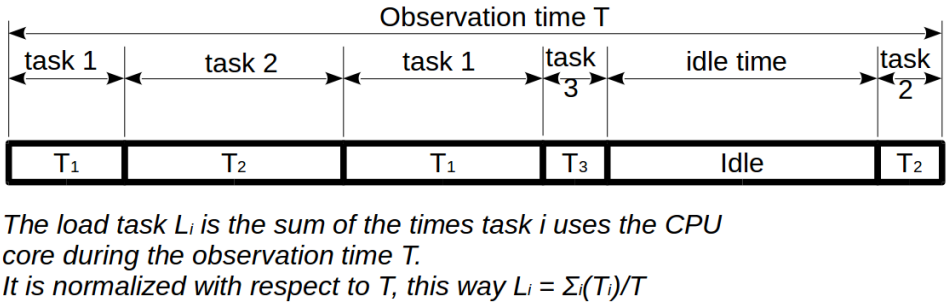
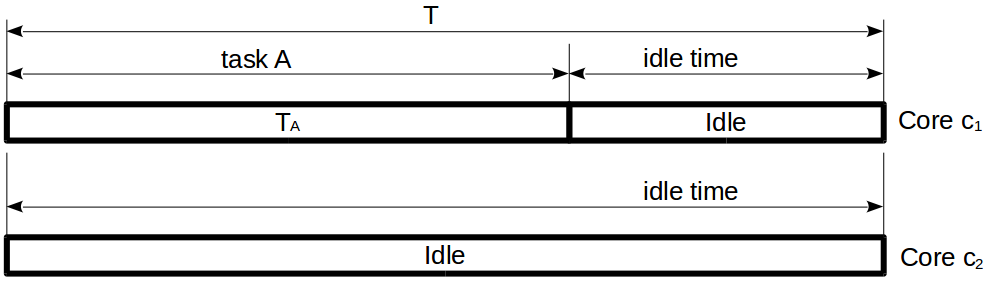


Figure 9: Task Load definition: Tasks schedule in one CPU core in one slice of time T.

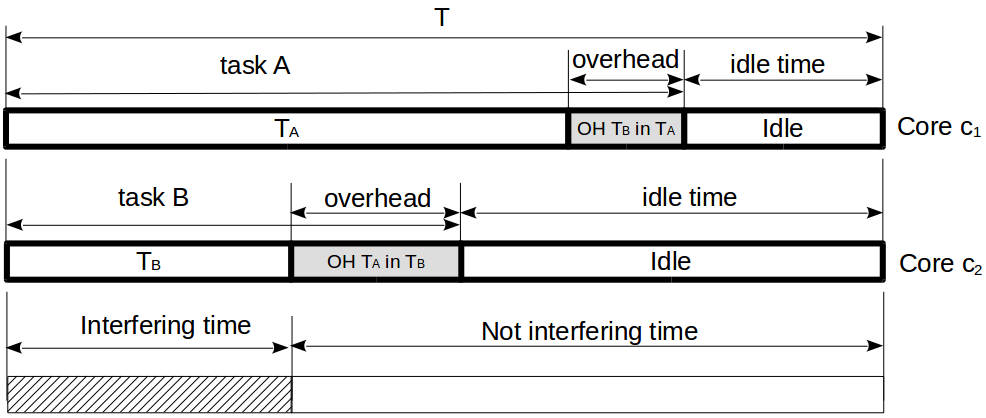
and is defined as the sum of total loads of all CPU cores in the system; $L_{sys} \in [0,N]$. The pipe notation $T_i|T_j$ denotes that the tasks T_i and T_j are running in different CPU cores. Using pipe notation we have $L_{sys}(A|B)$ being the normalized system load when task A is running in one CPU core and task B in another CPU core.

$$\mathrm{L}_{sys}(A|B) = L_A + L_B + L^{oh}_{A|B} \tag{18}$$

$L^{oh}_{A|B}$ is the parasitic load caused by task A running in one CPU core and task B running in another CPU core. This is maybe the most essential part of the model and reflects the waiting time where one CPU core has to stop the execution of a task because of a busy common resource, extending the execution time. The idea is that this extended execution time is enough to model the interaction and make dependable predictions. In Figure 10 it is shown the interpretation of the $L^{oh}_{A|B}$, where it can be seen that the interaction is only present when there are tasks running in parallel on both CPU cores. The formal definition



(a) Task A running on CPU core c_1, with no interference of CPU core c_2



(b) Task A running on CPU core c_1 is interfered by task B running on CPU core c_2.

Figure 10: Interference between tasks running on different cores. (a) Task A running on CPU core c1, with no interference of CPU core c2, (b) Task A running on CPU core c1 is interfered by task B running on CPU core c2.

of $L^{oh}_{A|B}$ is derived by the equation 19,

$$\mathrm{L}^{oh}_{A|B} = \beta_{A|B} L_A L_B \tag{19}$$

where $\beta_{A|B}$ expresses the level of interference (or coupling) that task A running in one CPU core and task B running in another CPU core exert on each other. This interference is assumed to be constant and dependent only on the nature of the tasks itself. If two tasks have affinity to the same resources, is expected to be higher than if the tasks have different affinities. Equation 19 shows $L^{oh}_{A|B}$ as function of $\beta_{A|B}$ that represents the interference due to the nature of the tasks, and the product of the individual loads. The idea behind it is to map the amount of time where the interference actually works as the product of the individual loads.

One could think that β could make more explicit the internals of the processors and peripherals. It could be done considering it as a vector, and each component would refer

to one common resource, like a FPU (Floating Point Unit) that is shared with two cores or because the cores are hyper-threaded, same for ALU (Arithmetic and Logic Unit) and this analysis would be extended to caches, memory, disk, I/O, etc. The problem with it is that this vector would be like 40 or even 50 components long (TANG et al., 2007), it would greatly increase the complexity of the model.

In order to overcome this problem the overhead load function is decomposed in individual performance penalties related to each task,

$$\mathrm{L}_{A|B}^{oh} = \beta_{A\to B} L_A L_B + \beta_{B\to A} L_A L_B \tag{20}$$

where $\beta_{A\to B}$ represents the interference of task A over task B and $\beta_{B\to A}$ represents the interference of task B over task A.

The same formulation can be expressed in matrix notation, by Equations 21 and 22, or in shorten form by equation 23,

$$\beta_{A\to B} L_A L_B = \begin{bmatrix} L_A & 0 \end{bmatrix} \begin{pmatrix} \beta_{A\to A} & \beta_{A\to B} \\ \beta_{B\to A} & \beta_{B\to B} \end{pmatrix} \begin{bmatrix} 0 \\ L_B \end{bmatrix} \tag{21}$$

$$\beta_{B\to A} L_B L_A = \begin{bmatrix} 0 & L_B \end{bmatrix} \begin{pmatrix} \beta_{A\to A} & \beta_{A\to B} \\ \beta_{B\to A} & \beta_{B\to B} \end{pmatrix} \begin{bmatrix} L_A \\ 0 \end{bmatrix} \tag{22}$$

$$L_{1\to 2}^{oh} = \mathbf{T}_1^T \mathbf{B} \mathbf{T}_2 \tag{23}$$

where $\mathbf{B}$ is the matrix of β-coefficients, $\mathbf{T}_1$ and $\mathbf{T}_2$ are vectors that represents the task running on CPU core 1 and 2, respectively, and $L_{1\to 2}^{oh}$ is the overhead that the tasks running on CPU core 1 impose to the tasks running on CPU core 2. This matrix representation is very useful because usually the CPU cores are running threads that sometimes have the same kind of task, specially in some kind of servers, like database, web-services or specialized services. In this matrix formulation the diagonal has the interference effect of threads of one kind of task over threads of same kind but running in different CPU cores.

A further generalization can be formulated to calculate the pairwise overhead with the

two CPU cores running a multitude of tasks, lets say n CPU cores and m tasks. Equations 24 and 25 show this generalized formulation,

$$\begin{aligned} L^{oh}_{sys} &= L^{oh}_{c1\to c2} + L^{oh}_{c1\to c3} + ... + L^{oh}_{c1\to cn} \\ &+ L^{oh}_{c2\to c1} + L^{oh}_{c2\to c3} + ... + L^{oh}_{c2\to cn} \\ &+ ... + L^{oh}_{cn\to c1} + L^{oh}_{cn\to c2} + ... + L^{oh}_{cn\to cn-1} \\ &= \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j\neq i}}^{n} \mathbf{T}_i^T \mathbf{B} \mathbf{T}_j \end{aligned} \tag{24}$$

where the overhead imposed on CPU core ci, running m tasks from $t1$ to tm, by the interference of CPU core cj, running the same tasks, is calculated by:

$$\begin{aligned} L^{oh}_{ci\to cj} &= \beta_{t1\to t2} L^i_{t1} L^j_{t2} + \beta_{t1\to t3} L^i_{t1} L^j_{t3} + ... + \beta_{t1\to tm} L^i_{t1} L^j_{tm} + \\ &\beta_{t2\to t2} L^i_{t1} L^j_{t2} + \beta_{t2\to t3} L^i_{t2} L^j_{t3} + ... + \beta_{t2\to tm} L^i_{t2} L^j_{tm} + \\ &\beta_{tm\to t2} L^i_{tm} L^j_{t2} + \beta_{tm\to t3} L^i_{tm} L^j_{t3} + ... + \beta_{tm\to tm} L^i_{tm} L^j_{tm} \\ &= \begin{bmatrix} L^i_{t1} & L^i_{t2} & ... & L^i_{tm} \end{bmatrix} \begin{pmatrix} \beta_{t1\to t1} & \beta_{t1\to t2} & ... & \beta_{t1\to tm} \\ \beta_{t2\to t1} & \beta_{t2\to t2} & ... & \beta_{t2\to tm} \\ ... & ... & ... & ... \\ \beta_{tm\to t1} & \beta_{tm\to t2} & ... & \beta_{tm\to tm} \end{pmatrix} \begin{bmatrix} L^j_{t1} \\ L^j_{t2} \\ ... \\ L^i_{tm} \end{bmatrix} \end{aligned} \tag{25}$$

Finally the estimated performance of the system (meaning a particular CPU core ci) can be seen as its effective load L_{sys}, that is calculated as the subtraction of the ideal system load (the load when there is no concurrency) by the system overhead calculated by Equation 25. The overhead represents a "penalty" on the ideal system load caused by the interference due to the contention for shared resources. Equation 26 shows the effective load computation,

$$L_{sys} = \sum_{i=1}^{n} \mathbf{T}_i - \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j\neq i}}^{n} \mathbf{T}_i^T \mathbf{B} \mathbf{T}_j \ . \tag{26}$$

For our purposes BFO is better suited than the high level models or the cycle-accurate models, because it describes in only by the use of single parameter β the interference between the cores of a processor, speeding up the performance prediction calculations. It is fundamental if one wants to predict the performance behavior of a processor for other purposes than prototyping, or benchmarking processors. For example, if someone wants to predict the performance of a computational system with commercial processors running an commercial application, with the interference of other applications running in parallel. For example,

a personal computer running several services in parallel, or a server running administrative services in parallel to the core application.

4 EXTENSIONS ON CLOUDSIM TO IMPLEMENT THE BFO MODEL

4.1 THE CLOUDSIM

CloudSim is a well known simulator used in industry and in academia as well. According to the CloudSim creator, this software is "*...an extensible simulator toolkit that enables modeling and simulation of Cloud computing systems and application provisioning environments.*". It is an open source Java based programming framework that implements an API (Application Programming Interface) where a user is free to extend the classes if needed. The simulation is based on a creation of a executable class (with main method) that implements the logic that is under study. CloudSim is suitable to simulate the most common areas of research like energy efficiency, performance/QoS, and as already said it can be extended to cover other areas as well. As open source the authors only asks to cite their work if you used and/or extended the simulator in your own research.

4.1.1 CLOUDSIM DESIGN

The CloudSim is originally designed to represent in software level the abstraction of a Cloud system. Figure 11 shows a Cloud layered design that is the basis for the CloudSim implementation. However, in our extensions, these services will not use as originally designed for, instead it will be used as a platform to run our jobs and check the CPU performance scaling against the measured performance. Therefore, the only levels we will touch are the System Level, that is where the physical and virtual resources are modeled, and the User Level, that is where the scenario to be simulated is implemented. For our purposes, the virtual resources with physical resources are tied together in a way that they are indistinguishable. This is a must in our stage of development, because in real clouds the virtual resource is constantly changing, it would make our efforts to validate the model too difficult if not impossible.

In the *System Level* we have the so called Cloud Resources, that are the computational resources like compute nodes (named Hosts) and the CPUs and vCPUs (virtual CPUs), RAM

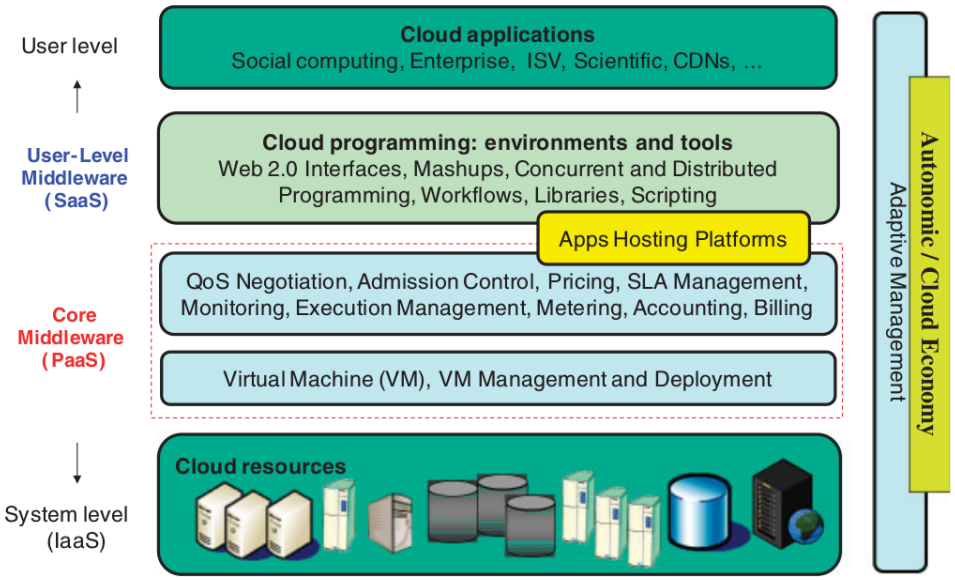


Figure 11: CloudSim Layered Design
Source: (CALHEIROS et al., 2009)

and vRAM (virtual RAM), Disk and vDisk (virtual Disk) that compose it. Hosts can be optimized to deliver computational power, storage or networking, so in the CloudSim we will have Hosts specialized in each of these functions. This level represents the classes and abstractions for the IaaS (Insfrastructure as a Service) service model. We adopted CloudSim release 3.2 more specificaly the extension called *DVFS CloudSim* (GUÉROUT et al., 2013), it was developed by Guérout et al. to implement DVFS (Dynamic Voltage and Frequency Scaling). This implementation is used because it implements the basis of TLM (Transaction-Level Modeling) simulation, that allows the recalculation of the state of every element every tick of the simulation clock.

The *User Level* represents the user applications, represented as tasks to be consumed. CloudSim uses the Cloudlet class to abstract the application modeling it in terms of a size in bytes a task represents, this Cloudlet will be consumed in a VM objects that have an amount of vRAM, vCPU, and vDisk, those VMs are instantiated in Hosts objects that contain the description of physical CPU, RAm and disk. To avoid a vCPU change the physical CPU during the runtime, we fixed the scheduler to pic and fix a pair vCPU-CPU in a Host. Moreover, to automatize the modeling, we introduced an XML based description archive to describe the data-center, and its components (Hosts, VMs, Cloudlets) in terms of its components. The details of this implementation will be shown in the next sections.

4.1.2 CLOUDSIM ARCHITECTURE

A brief description of the CloudSim API will help to understand the implementation. It is a Java based API, modeled as extensible classes that can be modified in a reasonable time to match the necessities of a researcher, notably for modeling VM allocation, modeling Cloud Market, modeling Network Behavior, modeling a Federation of Clouds, modeling Data Center Power Consumption and modeling Dynamic Entities Creation.

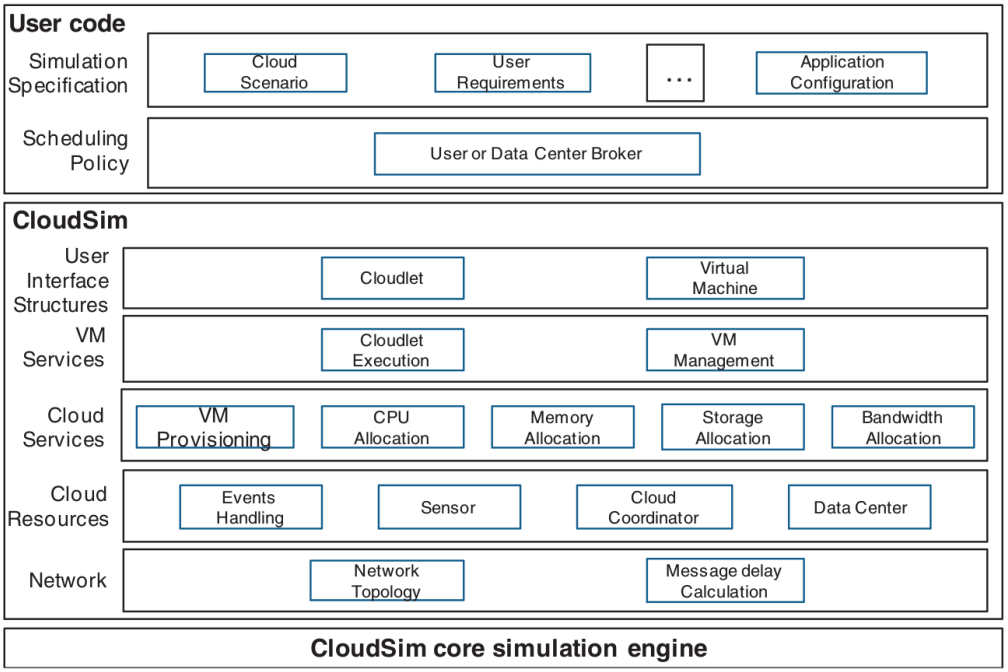


Figure 12: CloudSim Layered Architecture.
Source: (CALHEIROS et al., 2011)

CloudSim has a layered architecture that can be seen in Figure 12. This model shows an hierarchical software structure where one layer is client of the bellow layer and provides a service for the layer on its top. There are three main layers: User Code, CloudSim and CloudSim core simulation engine. The User Code is subdivided in Simulation Specification and Scheduling Policy. CloudSim is subdivided in User Interface Structures, VM Services, Cloud Services, Cloud Resources and Network. How these layers and entities workout together is described bellow.

The User code is the space where the researcher/user can write a code to simulate the scenarios under test. The user creates a code based on the simulation specifications, making use of the classes made available by CloudSim layer. To facilitate and accelerate the programming,

User Code layer makes available the *DataCenterBroker* class, it represents the mediating entity responsible for the negotiations (for example an application call to be executed) between SaaS (Software as a Service) and Cloud Providers.

The CloudSim layers provides the abstracted and extensible classes that represents the building blocks of a Cloud system. The User Interface Structures contains the *Cloudlet* and the *Vm* (Virtual Machine) classes. The Cloudlet class provides an abstracted model for the application demands in terms of quantity of computational resources required like how many MIPS (Million Instructions Per Second), how much memory and size of the archive to be processed. The *Vm* class provides an abstracted model for a Virtual Machine in terms of its physical resources demanded (the usually named *flavor*). The *Vm* resources are CPUs, MIPS, memory size, disk size (storage) and network bandwidth. The *Vm* Services layer contains the life-cycle management of the *Vm* and *Cloudlet* entities, it is done via the *Cloudlet* Execution and *VMM* (VM management) classes. The Cloud Services layer provides the abstracted and some already implemented classes to support the *VMM* and the *Cloudlet* Execution, these services are related to allocate a Host that have available resources to a *Vm* to run and once it allocated make the queued *Cloudlet*'s calculate the estimated execution time other elements like energy consumption and costs. On the other hand once the *Cloudlet*'s queue become empty the *Vm* is deallocated finishing the life-cycle. CloudSim also supports the pause/resume of the *Cloudlet*'s.

4.1.3 CLOUDSIM CLASSES

Figure 13 shows the class design diagram of CloudSim. The diagram does not shows all the components in the CloudSim Package that someone can download, but it has the fundamental parts that build the simulator. It will be described the ones touched in our implementation, following the CloudSim documentation (CALHEIROS et al.,).

- *Cloudlet:* It is an implementation to the application modeling. It stores, along with all the information encapsulated in this class, the ID of the *Vm* that is running it. This class models the Cloud-based application services (such as content delivery, social networking, and business work-flow). CloudSim orchestrates the complexity of an application in terms of its computational requirements. Every application service has a pre-assigned instruction length and data transfer (both pre and post fetches) overhead that it needs to undertake during its life cycle. This class can also be extended to support modeling of other performance and composition metrics for applications such as transactions in database-oriented applications.

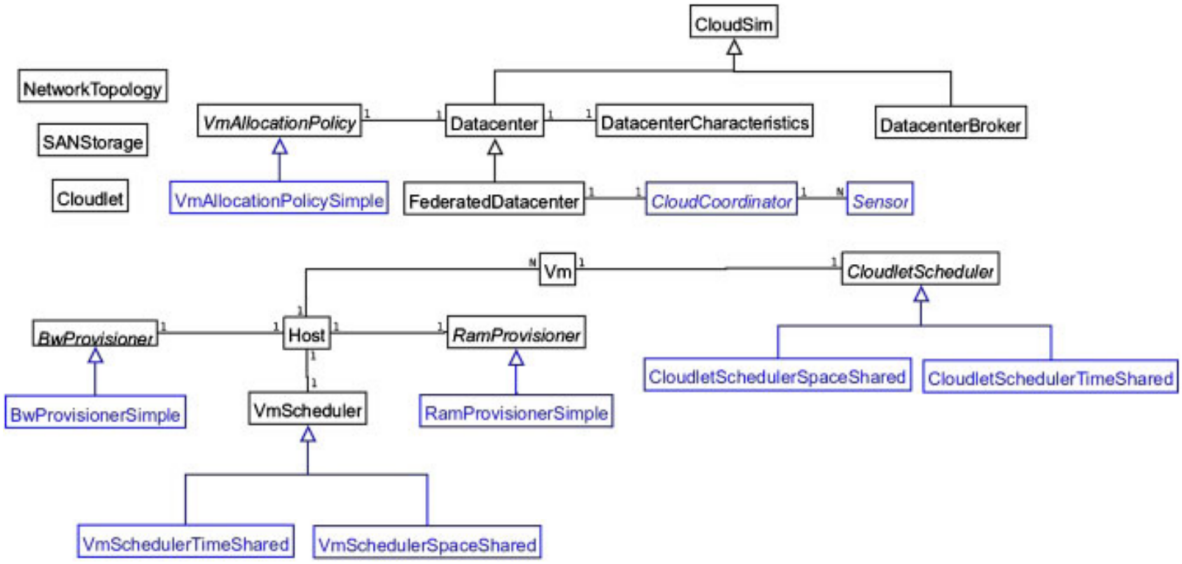


Figure 13: CloudSim Layered Architecture
Source: (CALHEIROS et al., 2011)

- *CloudletScheduler:* It is an abstract class that represents the policy of scheduling performed by a virtual machine. So, classes extending this must execute *Cloudlet*'s. Also, the interface for *Cloudlet* management is also implemented in this class.
  This abstract class is extended by the implementation of different policies that determine the share of processing power among *Cloudlet*'s in a *Vm*. Two types of provisioning policies are offered: space-shared (*CloudetSchedulerSpaceShared*) and time-shared (*CloudletSchedulerTimeShared*).

- *Datacenter:* This class is a CloudResource whose *hostList* is virtualized. It deals with processing of *Vm* queries (i.e., handling of VMs) instead of processing *Cloudlet*-related queries. So, even though an AllocPolicy will be instantiated (in the init() method of the superclass, it will not be used, as processing of cloudlets are handled by the *CloudletScheduler* and processing of Virtual Machines are handled by the *VmAllocationPolicy*. This class models the core infrastructure-level services (hardware) that are offered by Cloud providers (Amazon, Azure, App Engine). It encapsulates a set of compute hosts that can either be homogeneous or heterogeneous with respect to their hardware configurations (memory, cores, capacity, and storage). Furthermore, every *Datacenter* component instantiates a generalized application provisioning component that implements a set of policies for allocating bandwidth, memory, and storage devices to hosts and VMs.

- *DatacenterCharacteristics:* It represents static properties of a resource such as resource

architecture, Operating System (OS), management policy (time- or space-shared), cost and time zone at which the resource is located along resource configuration.

- *Host:* It executes actions related to the management of virtual machines (e.g., creation and destruction). A host has a defined policy for provisioning memory and network bandwidth, as well as an allocation policy for Processing Elements *Pe*'s (CloudSim class that models the vCPUs) to the virtual machines. A host is associated to a datacenter. It can host virtual machines. This class models a physical resource such as a compute or storage server. It encapsulates important information such as the amount of memory and storage, a list and type of processing cores (to represent a multi-core machine), an allocation of policy for sharing the processing power among *Vm*'s, and policies for provisioning memory and bandwidth to the *Vm*'s.

- *Vm:* This class represents a virtual machine: It runs inside a *Host*, sharing *hostList* with other *Vm*'s. It processes cloudlets. This processing happens according to a policy, defined by the *CloudletScheduler*. Each *Vm* has an owner, which can submit cloudlets to the *Vm* to be executed. This class models a virtual machine, which is managed and hosted by a Cloud host component. Every *Vm* component has access to a component that stores the following characteristics related to a virtual machine: accessible memory, processor, storage size, and the *Vm*'s internal provisioning policy that is extended from an abstract component called the *CloudletScheduler*.

- *VmmAllocationPolicy:* It is an abstract class that represents the provisioning policy of hosts to virtual machines in a Datacenter. It supports two-stage commit of reservation of hosts: first, we reserve the host and, once committed by the user, it is effectively allocated to he/she. This abstract class represents a provisioning policy that a *Vm* Monitor utilizes for allocating *Vm*'s to hosts. The chief functionality of the *VmmAllocationPolicy* is to select the available host in a data center that meets the memory, storage, and availability requirement for a *Vm* deployment.

- *VmScheduler:* It is an abstract class that represents the policy used by a *VMM* to share processing power among *Vm*'s running in a host. This is an abstract class implemented by a *Host* component that models the policies (space-shared, time-shared) required for allocating processor cores to *Vm*'s. The functionalities of this class can easily be overridden to accommodate application-specific processor sharing policies.

4.1.4 DATACENTER INTERNAL PROCESSING

Figure 14 shows a sequence diagram with the *Cloudlet* processing update process. In each simulation step the task units (*Cloudlet*'s) must be consumed in the respective *Vm*

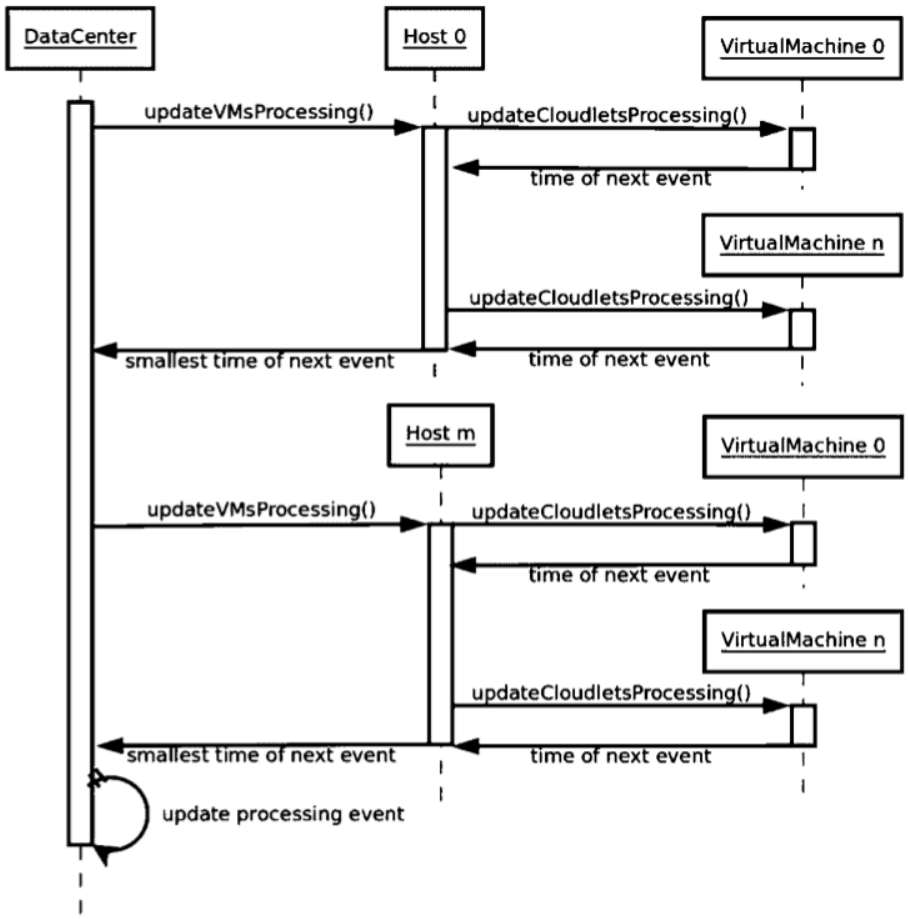


Figure 14: *Cloudlet* Processing Update Process.
Source: (CALHEIROS et al., 2011)

scheduled to treat this task. It is done step by step until no *Cloudlet*'s are left in the list of *Cloudlet*'s. To make it work, the *Datacenter* entity receives an internal event informing that a task unit completion is needed. The *Datacenter* entity invokes a method called *updateVMsProcessing()* for each *Host* in the list of *hosts* it manages. Each *Host* upon receiving this request contacts the *Vm* in its own list of *Vm*'s invoking the *updateCloudletProcessing()*. Each *Vm* answers the Host with the task unit status (finish, suspended, executing) with the *Datacenter* entity. Each *Vm*'s in the list return the next expected completion time, and the least completion time among all is then sent to *DataCenterEntity*. This value is used to progress the simulated clock to the next tick (one tick is equal the least completion time among all computed values). This sequence is repeated until the last *Vm* compute the last task unit, then the clock

freezes and the simulation stops.

The CloudSim simulation cycle is represented in the sequence diagram of Figure 15. It all starts with the registration of all *Datacenter*'s in CISRegistry entity, this way CIS

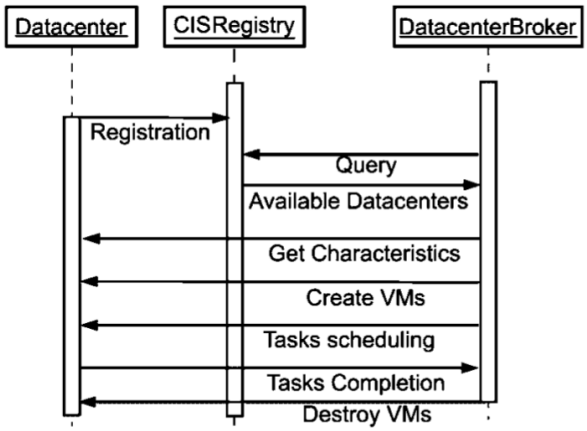


Figure 15: *Cloudlet* Simulation Cycle.
Source: (CALHEIROS et al., 2011)

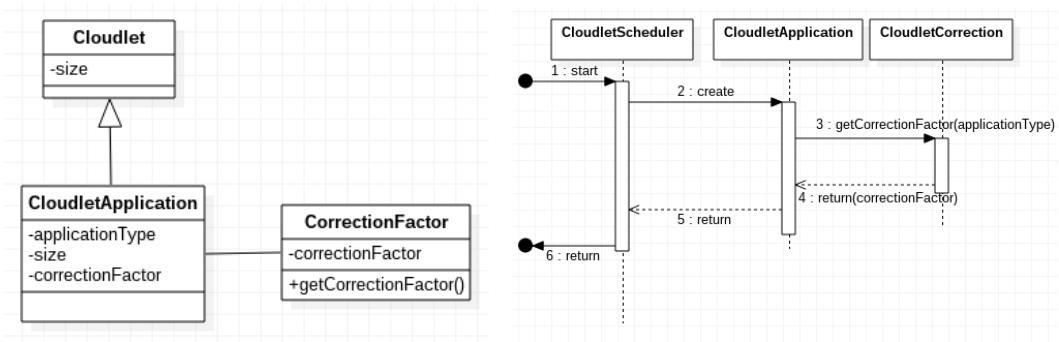
(Cloud Information System) has the information to deliver to brokers of what *Datacenter*'s are available. Once a user decides to deploy *VM*'s, it calls the *DatacenterBroker* that queries *CISRegistry* about the *Datacenter*'s available, receiving a list of available *Datacenter*'s. *DatacenterBroker* then queries the *Datacenter*'s to retrieve their characteristics (prices, power consume, virtualization technology, number of hosts, number of CPUs, vRAM, vDisk, and so on). The information is then used by *DatacenterBroker* to deploy the *Vm* on behalf of the user, based on pre-implemented schedulers or via a customized scheduler, it is also true about the *Cloudlet*'s (also known as task unit) that are scheduled to be run in the *VM*'s or *Cloudlet*'s. Once the *Datacenter* advances one tick of the clock it recalculates the state of the execution until the completion of all *Cloudlet*'s, when it informs *DatacenterBroker* that the *Cloudlet*'s were all consumed. All the reports and logs can be processed or post-processed and then the *DatacenterBroker* destroys all *Vm*'s created.

4.2 FIRST ORDER BEHAVIORAL MODEL IMPLEMENTATION IN CLOUDSIM

4.2.1 CLOUDLET CLASS EXTENSION

In CloudSim, CPUs are modeled by its capacity of consumption of the *Cloudlet*'s only based in immutable MIPS. It is not enough, because when we have different tasks, in different workflows, the same CPU has different performance (in other words, has different

capacity in MIPS). To work with this complexity we incorporate in the *Cloudlet* class the attributes named *applicationType* and *correctionFactor*. It permits that when the simulation consumes a *Cloudlet*, the CPU model can adapt to the type application it is running and perform accordingly to the previous measurements conducted in real systems. This implementation is shown in Figures 16(a) and 16(b). Where the original *Cloudlet* class is extended to have the attributes *applicationType*, that stores the name of the application a *cloudlet* object is carrying, and *correctionFactor*, that will be used to correct the *cloudlet* object size to be consumed by the CPUs. The *correctionFactor* is a table that we constructed in a helper class, the *CorrectionFactor* class. These tables with the *correctionFactor* are previously calculated in the real world CPUs for that particular application. See section 5.1.4.1 for details on the calculations of the *correctionFactor*.



(a) *CloudletApplication* as an extension of *Cloudlet* class

(b) Constructor of *CloudletApplication* asks for *correctionFactor* during instantiation of an object

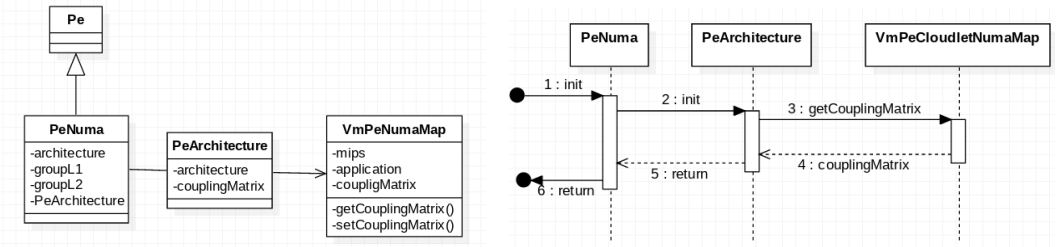
Figure 16: Extensions on Cloudlet class and object instantiation helped by *CloudletCorrection* class.

For our implementation the CloudSim simulator chosen was the *CloudSim-3.0.3* (GROZEV, 2012). Because it improved the control of the simulation advance compared with the former versions.

4.2.2 *PE* CLASS EXTENSION

The *Pe* Class was extended to accommodate the new CPU model, it was included the name of the real world CPU. This is important because this information need to be retrieved during simulation to apply the correct β that is dependent on the CPU type and the application type. For it in runtime, *Datacenter* class asks for the method *host.updateVmsProcessing*, the Class *PowerHost* will ask for the CPU model, that now return the MIPS corrected by the interference model. The new extended class is named *PeNuma* that instanciate an object of the class *PeArchitecture* that implements the interference calculation. The *PeArchitecture* class constructor uses a helper class named *VmPeCloudletNumaMap* to implement the correct

interference matrix based on the vector (CPU type, application type, NUMA zone), besides NUMA zone is part of the original design, it was not implemented yet. Figure 17(a) and 17(b) shows the extended classes, new helper classes and the relations they have.



(a) *PeNuma* as an extension of *Pe* class

(b) Constructor of *PeNuma* instanciate a *PeArchitecture* object that asks for the interference model (*couplingMatrix*) in the helper *VmPeNumaMap*

Figure 17: Extensions on *Cloudlet* class and object instantiation helped by *CloudletCorrection* class.

4.2.3 *POWERHOST* CLASSES OVERWRITING

To pin the vCPUs into CPUs (*PeNUma* objects), instead of checking the list of *Pe*'s at each tick of the simulator clock, we overwrite this method to maintain the original association when the *VmList* was originally created by the scheduler. This is done in the method *updateVmsProcessing*. Moreover, the *PeList*, was overloaded to accommodate the new *Pe* model (*PeNuma* class).

4.2.4 DATACENTER MODELED BY XML FILE

To make the simulations the most flexible possible, the datacenter object constructor is fed by the helper class *SimulationXMLParse*, that parses the parameters read from the file Experience.xml. The file models Hosts, CPus, Vms and Cloudlets that will be used in the simulation. Bellow we see some extracts of the file, to understand the modeling.

$< simulation >$

$< datacenters >$

$< datacenter >$

$< arch > x86 < /arch >< os > Linux < /os >< vmm > Xen < /vmm >$

$<!--ProcessingHostCreation-->$

$< host >$

$< ram > 10000 < /ram >< storage > 1000000 < /storage >< bw > 300000 < /bw >< maxP > 250 < /maxP >$

$< staticPP > 0.7 < /staticPP >$

<cpus><cpu>

<num>1</num><governor>performance</governor><architecture>i7</architecture>

<numaZone>0</numaZone>

</cpu>

<cpu>

<num>2</num><governor>performance</governor><architecture>i7</architecture>

<numaZone>1</numaZone>

</cpu>

<mips>1550.14</mips></host>

</datacenter></datacenters>

<vms><vm><mips>1550.14</mips><cpu>4</cpu><ram>128</ram><bw>2500</bw>

<size>2500</size>1550.14<vmm>Xen</vmm></vm></vms>

<cloudlets><cloudlet><length>347027.4</length><pes>4</pes>

<application>int128</application></cloudlet></cloudlets>

</simulation>

5 FRAMEWORK FOR MULTI-CORE SYSTEMS EVALUATION

The objective of this chapter is to present a methodology to get the parameters used to feed the simulations, as well as, the methodology to validate the simulators predictions with actual measured behavior collected on commercial systems. For that, we will divide the work in three phases:

1. First is the *testing phase*, where will be used real world collected data to calculate the coupling factors used in the BFO model. For that we will use some performance and benchmarking tools to read from the CPU counters and manipulate it following the method described in the section *Overhead Measurements*.

2. Second is the *training phase*, where the coupling factors and the adjusts of the real CPU power (in MIPS) are introduced in the simulator. It is used to reproduce the real world measured scenarios, but with one CPU only (meaning no concurrency). This is to ensure the CPU model is well done and can reproduce the results measured for the so called *ideal* scenarios.

3. Third is the *simulation phase*, this phase will be subdivided in two parts: the simulation to *reproduce* simple scenarios already tested during the testing phase and the simulations to *generalize* the simulator, showing it can be used to simulate new scenarios. The generalization simulations will be compared with tests in the real systems.

5.1 OVERHEAD MEASUREMENTS

To use the model it is necessary to acquire by experiments the coupling factor β. The methodology adopted to acquire the data necessary to calculate them is described bellow:

1. The hardware performance counters (HPC) statistics will be used. They are available in all OSs and are accessible by tools like *perf*, that is part of most of Linux distributions, or *System Monitor* in case of Windows OS. The HPC are special registers that collect

hardware statistics on runtime and have almost no overhead associated. According to Yilmaz (YILMAZ, 2010) these registers are extensively used for program debugging and profiling. In this work we will work with *perf* tool. This restriction is acceptable because according to Weaver et. al. (WEAVER; DONGARRA, 2010), the accuracy of HPC does not dependent on the OS and measuring tools, so it can be generalized even when other OSs are being used.

2. The basic metric that will be used is the *number of instructions per clock cycle* (I_c). This metric shows naturally the problem of the interference when a CPU core is waiting to use an already busy resource, or if it is fetching data that is not in the local cache. This last reason is really important one, because caches are much faster than accessing the local memory that in its turn is faster than accessing remote (i.e. in other NUMA [1] zone) memory. The effects of these factors is a stalling of the CPU core, that looses CPU clocks to wait for the busy resource or the data to be available.

   As any measure, the HPC are not exact and have some degree of uncertainty, the main reason is that these counters have to be collected on-line, so it uses resources to be processed and stored. According to Zaparanuks et al. (ZAPARANUKS et al., 2009) this accuracy is dependent of the number of enabled counters being collected and managed.

   However, this overhead in collecting and treating the counters is not the only source of error, another one is due to the interruptions that CPU core treat during runtime, the result is a possible duplication in the count of the interrupted instruction. This is always a over-count leading to overestimation of performance, but luckily it is small and sometimes partially compensate the underestimation caused by the overhead in collecting the data. Waver et al. (WEAVER; DONGARRA, 2010) estimates it to be less than 1% what is good for most of the practical purposes.

3. To train the model, i.e. to obtain the coupling factors β, first we will run multiple times (say 30 times) each of the tasks chosen, based in math intensive (specially matrix), video compression (that will use disk and GPU (Graphics Processing Unit)), some combination of stress tests (using *stress* tool to capture some uses (disk intensive, RAM intensive, i/o intensive or CPU intensive). It will give the "ideal performance" that will be used to compare with the performance with interference of activities in parallel. The second step is to run the same tasks, but now launching the task A in one CPU core and the other task B in another CPU core.

4. The methodology of measurement: The tasks will be launched using the *perf* tool, and

[1] NUMA (Non-Uniform Memory Access) is a kind of architecture for shared memory.

Table 1: Usage of taskset pinning flag. The flag is the hexadecimal interpretation when OFF = 0 and ON = 1.

CPU 3	CPU 2	CPU 1	CPU 0	Taskset flag
OFF	OFF	OFF	ON	0x1
OFF	OFF	ON	OFF	0x2
OFF	ON	OFF	OFF	0x4
ON	OFF	OFF	OFF	0x8

the result will be the average of 30 runs. Moreover to pin the task to a particular CPU core, the tool used will be *taskset*, that can be concatenated with *perf*.

5. When using the (extended) CloudSim to analyse a given multi-core system we used the root mean square error (RMSE), showed in Equation 27, to evaluate the quality of the predictions. The RMSE is given by:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\frac{I_c^{pred}(i) - I_c^{meas}(i)}{I_c^{meas}(i)}\right)^2} \tag{27}$$

where $I_c^{pred}(i)$ is the i^{th} value of I_c predicted by the extended CloudSim and $I_c^{meas}(i)$ is the i^{th} value of I_c measured in commercial systems, a low $RMSE$ means better PREDICTION and n is the number of runs during the validation phase.

5.1.1 HOW TO EMPIRICALLY OBTAIN THE COUPLING FACTORS β

In order to obtain the pairwise coupling factors β needed to the simulations, we will consider two distinct moments of measurements, the first one where a stressor is issued alone in one CPU core and no other activity is issued any other core (we will call it *Phase 1*). The second moment (the *Phase 2*) is when stressors are issued synchronously in a pair of cores, these stressors can be distinct ones or not.

Phase 1: In this phase we will use *taskset* to pin the stressor to one core, and use *perf* to read the number of instructions, named here X_i^j, with $i \in \{0,1,2,3,...\}$ being the number of the core in the processor and j = {A, B, C, D} is the stressors available to test represented here by letters. As the stressors are issued via *stress-ng* for a pre-defined amount of time Δt (we used $\Delta t = 320s$), we define $x_i^j = X_i^j/\Delta t$ as the capacity of core i to consume the stressor j, we will use the dimensional $[mips/s]$ for x_i^j. As the stressor is issued for 100% CPU power, for definition of BFO model, the task load $L_{sys} = 1$, and in this case $L_{i,j}^{full} = 1$, meaning that for

core i running stressor j the full utilization is reached. This test is done in all cores and with all stressors.

Now we can compare what happens when comparing these measurements pairwise, in other words, what should be the behavior of the system if no interference exist pair-wisely. In this case $L_{sys} = 2$, for definition, and we can say, without loss of generality, that for cores 0 and 1 and stressors A and B,

$$L_{sys} = L_{0,A}^{full} + L_{1,B}^{full} = 2 \ , \tag{28}$$

however, in general $X_0^{A,phase1} \neq X_1^{B,phase1}$ and $L_{0,A}^{full} = L_{1,B}^{full} = 1$, what means that the total number of instructions $X_0^{A,phase1} + X_1^{B,phase1}$ have different weights for stressor A and B. We define ε as the instruction ratio, as a representation of the weight that $X_0^{A,phase1}$ and $X_1^{B,phase1}$ has in the total of instructions available with no interference considered.

$$\varepsilon_A = \frac{X_0^{A,phase1}}{X_0^{A,phase1} + X_1^{B,phase1}} \quad and \quad \varepsilon_B = \frac{X_1^{B,phase1}}{X_0^{A,phase1} + X_1^{B,phase1}} \ . \tag{29}$$

As the time of test is fixed ε can also be calculated in terms of the core capacity x,

$$\varepsilon_A = \frac{x_0^{A,phase1}}{x_0^{A,phase1} + x_1^{B,phase1}} \quad and \quad \varepsilon_B = \frac{x_1^{B,phase1}}{x_0^{A,phase1} + x_1^{B,phase1}} \ . \tag{30}$$

<u>*Phase 2:*</u> The *perf* tool will present the total instructions that stressor A and B consumed to be completed. Because of the definitions of BFO model, we would need the instructions consumed in core 0 and core 1 individualized. To calculate it we assumed that ε is constant if the task load is unchanged, that is our case because stress-ng still consumes 100% of the core processing power. Defining $Z_{sys}^{meas.}$ as the number of instructions read in *perf* for *Phase 2*, we have:

$$X_0^{A,phase2} = \varepsilon_A . Z_{sys}^{meas.} \quad and \quad X_1^{B,phase2} = \varepsilon_B . Z_{sys}^{meas.} \ , \tag{31}$$

symilarly, using x,

$$x_0^{A,phase2} = \varepsilon_A . \frac{Z_{sys}^{meas.}}{\Delta t} \quad and \quad x_1^{B,phase2} = \varepsilon_B . \frac{Z_{sys}^{meas.}}{\Delta t} \ . \tag{32}$$

By definition, overhead OH is the part of the system processing that is spent with the

interference, in our case it is $OH = (X_0^{A,phase1} + X_1^{B,phase1}) - Z_{sys}^{meas.}$. In terms of task load,

$$L_{sys}^{oh} = \frac{OH}{Z_{sys}^{meas.}} = \frac{(X_0^{A,phase1} + X_1^{B,phase1}) - Z_{sys}^{meas.}}{Z_{sys}^{meas.}} \quad . \tag{33}$$

Now we have all to calculate β, by definition of overhead,

$$L_{sys}^{oh} = \beta . L_A^{0,phase2} . L_B^{1,phase2} \quad , \tag{34}$$

and we know that $L_A^{0,phase2} = L_B^{1,phase2} = 1$, that lead us to Equation 36 with the final form of β.

$$\beta = L_{sys}^{oh} = \frac{(X_0^{A,phase1} + X_1^{B,phase1}) - Z_{sys}^{meas.}}{Z_{sys}^{meas.}} \tag{35}$$

Finally, in the simulator, it is needed the individual contributions $\beta_{A\to B}$ and $\beta_{B\to A}$, that can easily be calculated as,

$$\beta_{A\to B} = \varepsilon_A . \beta \quad and \quad \beta_{B\to A} = \varepsilon_B . \beta \quad . \tag{36}$$

5.1.2 TESTING PROGRAMS AND HARDWARE CONFIGURATIONS

In the training and validation phases, we will use the *stress-ng* to generate the CPU activity of different profile to collect the data for the coupling factors. The choice for it is because it can generate controlled loads in various dimensions of the system, generating CPU load, RAM load, I/O load in a controlled way. The native load profiles, also known as *stressors*, chosen for this work are:

1. **INT128:** This stressor generates 1000 iterations of a mix of 128 bit integer operations. It is good for benchmarking the CPU performance for operations over integers.

2. **FFT:** This stressor generates blocks of 4096 samples for Fast Fourier Transform calculations. It is good for benchmarking the CPU performance for DSP (Digital Signal Processing) applications.

3. **MATRIXPROD:** This stressor generates matrix product of two 128 by 128 matrices of double floats. Testing on 64 bit x86 hardware shows that this provides a good mix of memory, cache and floating point operations and is probably the best CPU method to use to make a CPU run hot.

4. **CALLFUNC:** This stressor generates recursively call 8 argument C function to a depth of 1024 calls and unwind. It is good to benchmarking the memory scaling capabilities on

the processor cache.

The hardware used to make the measurements are:

1. Notebook with one processor Intel i7-6500U (2 individual cores, or 4 cores with hyper-thread [3] activated) , 16GB of RAM , 256 GB SSD (Solid State Disk) and 1GBps Ethernet network adapter.

2. Server Dell with two processors AMD Opteron (2 individual cores, no hyper-thread) , 32GB of RAM, 2TB HD and 4 1GBps Ethernet network adapter.

3. Server AirFrame Nokia with two processors XEON (24 individual cores, or 48 cores with hyper-thread activated), 256BG of DDR4 RAM, 512 GB SSD and 2x 6TB HD Raid 2.

5.1.3 TRAINING PHASE: RESULTS OF THE CALIBRATION TESTS

In the training phase the objective is to generate enough statistics in an experimental system in order to calculate the coupling factors β. The methodology to collect the statistics as well as the cases covered (the combinations of hardware and stressors) were discussed in earlier sections. Now we will present the results of the experimentation done in the training phase.

In Table 2 it is shown the number of instructions per second that each stressor generates for the ideal case, that is when only one CPU core is in use, and the background activity is minimum. This data is calculated as the average over 10 interactions on each CPU core for each run of the script. Considering a case where a processor have n CPU cores and the script will run for k stressors, the number of times the same statistic for instructions per second (m) is collected is $m = A_n^2.k^2.10$. For example, for 2 cores ($n = 2$), 4 stressors ($k = 4$) and 10 runs for each combination, we have $m = A_2^2.4^2.10 = 1.16.10 = 160$. This number is shown in the table as the row Number of Tests.

It is important to note that the value of MIPS measured varies in a fixed CPU type depending on each stressor. For example, consider the OPTERON CPU, for the stressor INT128 the value of measured MIPS is 1927.93, this contrasts with the FFT stressor that presented a value of 2163.79 MIPS. This is due to each stressor to use the resources in a

[3] Hyper-Threading is a technology where a single core is used to process two pipelines of instructions (threads), therefore it is perceived by the OS as if there were two active cores.

Table 2: Instructions per second (MIPS) calculated for each CPU type in the ideal case.

CPU / Stressor	INT128	FFT	MATRIXPROD	CALLFUNC
i7-6500U (MIPS)	1550.14	3204.71	4399.28	5289.63
Opteron 2214HE (MIPS)	1927.93	2163.79	1063.88	2190.07
XEON E5-2680v4 (MIPS)	1873.95	3990.38	5238.49	6168.87
Number of tests	160	160	160	160

particular way. The performance is also dependent to the CPU architecture and the pattern of the use of the resources. This information is very important because in CloudSim the capacity of a CPU (vCPU or pCPU) is defined in MIPS, and it is not dependent on the task to be simulated (named there Cloudlets), but if we are to simulate different kinds of task/applications, it is necessary to correct it.

Basically the correction is done by adjusting the size of the Cloudlet before the start of the simulation to maintain the time of execution coherent with the observed in the real world. For example, if a 2000 MIPS CPU is defined in CloudSim for a Host and VM objects, that represent a specific market CPU core, and this CPU has to consume a 10,000M instructions from INT128 and another 10,000M from CALLFUNC, if not corrected both tasks would finish in the same execution time of 5s. But, as we can see, the CPU would present different time of execution in the experimental world, it would finish CALLFUNC earlier then INT128. To illustrate it imagine that we were representing an i7-6500U CPU core, the task INT128 will be corrected to $INT128_{Corrected} = 10,000Mx2000/1550.14 = 12,902M$ instructions, and CALLFUNC to $CALLFUNC_{Corrected} = 10,000Mx2000/5289.63 = 3,781M$ instructions. So, the INT128 will finish in 6.451s and CALLFUNC in 1.89s. This correction will be important in the second phase that is the CloudSim extensions mimetizing the results in real world, and it is done during the instantiation on the Cloudlet when the correction factor is used, making it transparent to the user of the simulator. And the correction factors will be derived from the Table 2.

The results are shown in Table 3 for OPTERON, Table 4 for Xeon E5-2680v4 and Table 5 for i7-6500U. These tables are the inputs for the creation of the *CorrectionFactor* class in the extended CloudSim. For sake of simplicity, here we will create the class with the tables embedded, however, in a more professional approach it would be better implemented it in a XML archive. The class *CloudletApplication* than should import these data in runtime during

the object instantiation. This way the user just need to import the XML from a repository and use the credentials for the intended environment to be simulated.

Table 3: Coupling Factors $\beta_{A\rightarrow B}$ for OPTERON 2214HE

row:task A / column:taskB	INT128	FFT	MATRIXPROD	CALLFUNC
INT128	0.1973	0.553	1.2441	0.0698
FFT	0.6207	0.0487	0.0583	0.0524
MATRIXPROD	0.6865	0.0286	0.0536	0.00125
CALLFUNC	0.0792	0.0530	0.0026	0.2161

Table 4: Coupling Factors $\beta_{A\rightarrow B}$ for XEON E5-2680v4

row:task A / column:taskB	INT128	FFT	MATRIXPROD	CALLFUNC
INT128	0.2847	0.1792	0.0927	0.1140
FFT	0.3816	0.0752	0.2054	0.2228
MATRIXPROD	0.2592	0.2696	0.1376	0.2158
CALLFUNC	0.3754	0.3445	0.7054	1.1595

Table 5: Coupling Factors $\beta_{A\rightarrow B}$ for i7-6500U

row:task A / column:taskB	INT128	FFT	MATRIXPROD	CALLFUNC
INT128	0.3966	0.1785	0.1078	0.1265
FFT	0.3691	0.0192	0.2203	0.2101
MATRIXPROD	0.3059	0.3025	0.1287	0.2588
CALLFUNC	0.4317	0.3468	0.3112	2.0482

5.1.4 SIMULATION PHASE: VALIDATION OF EXTENSIONS MADE ON CLOUDSIM

After collecting and processing the data in the previous phase, the next step is validate the extended CloudSim to make sure it reproduces the results observed in the experimental systems. This phase will have two parts, the first one is to ensure that CloudSim reproduces the ideal results. In the second part we will apply the concept of interference overhead, to simulate the tests with two CPU cores interfering with one another. These two parts will ensure that CloudSim is capable to reproduce the BFO model validating the model and the implementation in software.

5.1.4.1 PART I: REPRODUCING THE IDEAL SCENARIOS

In this part, it is tested the concept that the *Cloudlet* must be corrected in its size, to reflect that the use of a CPU core from a particular task have differences inherent to it, as explained in Training Phase section. The methodology, for the tests to be done, is to create simulation environments mimetizing the real world systems, where cloudlets of different kind (*applicationType*) and sizes are created and executed. The simulated execution time will be the parameter to be compared with the real systems measured execution time.

Initially we will take the ideal tests scenario executed in the trainning phase and tweak the simulator parameters (in this case the *correctionFactor*) to calibrate the simulator. After this initial simulations, some cloudlets (bigger and smaller) will be simulated, and compared with the experimental measurements (over 10 runs). The quality of the results is based on the root mean square error showed in Equation 27. Table 6 shows the results of the tests for the stressors INT128 (Table 6(a)), FFT (Table 6(b)), MATRIXPROD (Table 6(c)) and CALLFUNC (Table 6(d)). The cloudlets sizes are 25%, 50%, 100%, 200% and 400% of the ideal scenarios. For RMSE analysis, the lower the better, so a RMSE = 0 would mean no difference between measured and simulated results. Following the same logic the Table 7(a),(b),(c) and (d) shows the results for Xeon and Table 8(a),(b),(c) and (d) shows the results for i7-6500U.

With these results we can see that the cloudlets can approximate the real world scenarios very well, as expected in these scenarios the execution time escalates linearly. Maybe it is not a very good approximation if the cloudlet is very short, since the model used has the assumption that the tasks/CPU cores are steady state. Meaning that the caches are already stabilized, since tasks too short and possibly uncoordinated can raise the memory access due to more often cache fails.

5.1.4.2 PART II- INTERFERENCE IN PAIRWISE SCENARIOS

Now the goal is to put CloudSim to simulate the scenarios that were used to calculate the coupling factors β. In this part what is to be tested is the extensions made in all classes to support the overhead calculations. This is important to check the model and the implementation, after that the next phase will treat with the situation where tasks will occupy all the cores in a VM, and then compare the performance of the simulated scenario with the experimental tests.

Table 6: CloudSim simulating various stressors results with Opteron 2214HE parameters (declared in CloudSim with MIPS=3000)

(a) Simulating Opteron 2214HE for INT128

INT128 Cloudlet	**25%**	**50%**	**100%**	**200%**	**400%**
size (Millions Inst.)	86756.85	173513.7	347027.4	694054.8	1388109.6
corrected size factor=1.5561	135000	270000	540000	1080000	2160000
CloudSim exec. time (s)	45.01	90.03	179.98	360.04	719.96
Real System exec. time (s)	47.55	90.03	180.41	359.52	718.05
Δ%	-5.34	-0.95	-0.24	0.14	0.27
RMSE (%)	0.0435	0.0225	0.0100	0.0102	0.0091

(b) Simulating Opteron 2214HE for FFT

FFT Cloudlet	**25%**	**50%**	**100%**	**200%**	**400%**
size (Millions Inst.)	97370.55	194741.1	389482.2	778964.4	1557928.8
corrected size factor=1.3865	135000	270000	540000	1080000	2160000
CloudSim exec. time (s)	45.00	89.99	180.04	359.97	719.99
Real System exec. time (s)	46.98	90.93	180.49	360.33	720.97
Δ%	-4.21	-1.03	-0.25	-0.10	-0.14
RMSE	0.0398	0.0194	0.0093	0.0062	0.0096

(c) Simulating Opteron 2214HE for MATRIXPROD

MATRIXPROD Cloudlet	**25%**	**50%**	**100%**	**200%**	**400%**
size (Millions Inst.)	47874. 6	95749.2	191498.4	382996.8	765993.6
corrected size factor=2.8199	135000	270000	540000	1080000	2160000
CloudSim exec. time (s)	45.02	90.02	180.05	360.08	720.03
Real System exec. time (s)	46.67	90.88	180.15	360,21	720.43
Δ%	-3.54	-0,95	-0,06	-0,04	-0,06
RMSE	0.0425	0.0234	0.0082	0.0110	0.0088

(d) Simulating Opteron 2214HE for CALLFUNC

CALLFUNC Cloudlet	**25%**	**50%**	**100%**	**200%**	**400%**
size (Millions Inst.)	98553.15	197106.3	394212.6	788425.2	1576850.4
corrected size factor=1.3698	135000	270000	540000	1080000	2160000
CloudSim exec. time (s)	44.98	90.03	180.08	360.07	720.11
Real System exec. time (s)	47.53	91.09	180.75	361.23	719.29
Δ%	-5.37	-1.16	-0.37	-0.32	0.11
RMSE	0.0460	0.0225	0.0133	0.0115	0.0073

Table 9 synthesizes the RMSE calculated over the simulation results against the pairwise measurements done in training phase for OPTERON, likewise Table 10 shows the RMSE when the processor is Xeon and Table 11 considers i7-6500U.

We observed that the error is acceptable, showing very good approximation with the

Table 7: CloudSim simulating various stressors results with Xeon E5-2680v4 parameters (declared in CloudSim with MIPS=6000)

(a) Simulating Xeon E5-2680v4 for INT128

INT128 Cloudlet	**25%**	**50%**	**100%**	**200%**	**400%**
size (Millions Inst.)	84327.75	168655.5	337311	674622	1349244
corrected size factor=3.2018	270000	540000	1080000	2160000	4320000
CloudSim exec. time (s)	45.03	90.02	180.01	360.03	719.94
Real System exec. time (s)	47.67	90.99	180.51	359.17	721.19
Δ%	-5.54	-1.07	-0.28	0.24	-0.17
RMSE	0.0435	0.0225	0.0100	0.0102	0.0091

(b) Simulating Xeon E5-2680v4 for FFT

FFT Cloudlet	**25%**	**50%**	**100%**	**200%**	**400%**
size (Millions Inst.)	179586	359172	718344	1436688	2873376
corrected size factor=1.5035	270000	540000	1080000	2160000	4320000
CloudSim exec. time (s)	45.01	90.01	180.00	359.96	720.03
Real System exec. time (s)	47.56	90.67	180.19	358.93	719.31
Δ%	-5.36	-0.73	-0.11	-0.29	0.10
RMSE	0.0453	0.0186	0.0062	0.0099	0.0062

(c) Simulating Xeon E5-2680v4 for MATRIXPROD

MATRIXPROD Cloudlet	**25%**	**50%**	**100%**	**200%**	**400%**
size (Millions Inst.)	235732.05	471464.1	942928.2	1885856.4	3771712.8
corrected size factor=1.1454	270000	540000	1080000	2160000	4320000
CloudSim exec. time (s)	44.99	90.02	180.03	359.99	720.08
Real System exec. time (s)	47.77	91.01	179.45	360.21	721.03
Δ%	-5.82	-1.09	0.32	-0.06	-0.13
RMSE	0.0520	0.0233	0.0106	0.0053	0.0068

(d) Simulating Xeon E5-2680v4 for CALLFUNC

CALLFUNC Cloudlet	**25%**	**50%**	**100%**	**200%**	**400%**
size (Millions Inst.)	277599.15	555198.3	1110396.6	2220793.2	4441586.4
corrected size factor=0.9726	270000	540000	1080000	2160000	4320000
CloudSim exec. time (s)	44.99	90.02	180.07	360.04	720.08
Real System exec. time (s)	47.83	90.89	179.84	359.23	719.41
Δ%	-5.94	-0.96	0.13	0.23	0.09
RMSE	0.0477	0.0208	0.0077	0.0088	0.0058

measured experiment. It is important to note that these tests are validating the BFO model and the CloudSim extensions as well, therefore the accuracy of the CloudSim is improved.

Table 8: CloudSim simulating various stressors results with i7-6500U parameters (declared in CloudSim with MIPS=5000)

(a) Simulating i7-6500U for INT128

INT128 Cloudlet	**25%**	**50%**	**100%**	**200%**	**400%**
size (Millions Inst.)	69756.3	139512.6	279025.2	558050.4	1116100.8
corrected size factor=3.2255	225000	450000	900000	1800000	3600000
CloudSim exec. time (s)	45.05	90.03	180.05	359.99	719.99
Real System exec. time (s)	47.73	91.09	180.25	359.07	720.93
Δ%	-5.61	-1.16	-0.11	0.26	-0.13
RMSE	0.0471	0.0214	0.0069	0.00104	0.0073

(b) Simulating i7-6500U for FFT

FFT Cloudlet	**25%**	**50%**	**100%**	**200%**	**400%**
size (Millions Inst.)	144211.95	288423.9	576847.8	1153695.6	2307391.2
corrected size factor=1.5602	225000	450000	900000	1800000	3600000
CloudSim exec. time (s)	44.99	90.03	180.04	360.04	719.97
Real System exec. time (s)	48.00	90.83	180.71	360.94	719.03
Δ%	-6.27	-0.88	-0.37	-0.25	0.13
RMSE	0.0457	0.0173	0.0128	0.0106	0.0077

(c) Simulating i7-6500U MATRIXPROD

MATRIXPROD Cloudlet	**25%**	**50%**	**100%**	**200%**	**400%**
size (Millions Inst.)	197967.6	395935.2	791870.4	1583740.8	3167481.6
corrected size factor=1.1365	225000	450000	900000	1800000	3600000
CloudSim exec. time (s)	45.00	90.02	180.07	360.01	720.02
Real System exec. time (s)	46.79	91.32	180.36	361.24	720.73
Δ%	-3.83	-1.42	-0.16	-0.34	-0.10
RMSE	0.0422	0.0245	0.0073	0.0110	0.0058

(d) Simulating i7-6500U for CALLFUNC

CALLFUNC Cloudlet	**25%**	**50%**	**100%**	**200%**	**400%**
size (Millions Inst.)	238033.35	476066.7	952133.4	1904266.8	3808533.6
corrected size factor=0.9452	225000	450000	900000	1800000	3600000
CloudSim exec. time (s)	45.09	90.07	180.03	360.04	720.07
Real System exec. time (s)	47.88	91.01	180.44	359.53	719.37
Δ%	-5.83	-1.03	-0.23	0.14	0.10
RMSE	0.0486	0.0219	0.0095	0.0075	0.0062

5.1.5 GENERALIZATION PHASE - USE OF EXTENDED CLOUDSIM TO PREDICT THE BEHAVIOR OF DIFFERENT LOADS SCENARIO

In this section the idea is to test the extended CloudSim in different loads scenarios, and compare the results with experimental measurements. In addition, the same scenario is set

Table 9: RMSE between CloudSim simulations over stressors and real world measurements with Opteron 2214HE. Tests calibrated for 320s of runtime.

core1/core2 (not siblings)	INT128	FFT	MATRIXPROD	CALLFUNC
INT128	0.0206	0.0174	0.0292	0.0075
FFT	0.0174	0.0142	0.0123	0.0368
MATRIXPROD	0.0292	0.0123	0.0355	0.0277
CALLFUNC	0.0075	0.0368	0.0277	0.0221

Table 10: RMSE between CloudSim simulations over stressors and real world measurements with Xeon E5-2680v4. Tests calibrated for 320s of runtime.

core1/core2 (not siblings)	INT128	FFT	MATRIXPROD	CALLFUNC
INT128	0.0305	0.0343	0.0022	0.0174
FFT	0.0343	0.0356	0.0166	0.0143
MATRIXPROD	0.0022	0.0166	0.0320	0.0192
CALLFUNC	0.0174	0.0143	0.0192	0.0381

Table 11: RMSE between CloudSim simulations over stressors and real world measurements with i7-6500U. Tests calibrated for 320s of runtime.

core1/core2 (not siblings)	INT128	FFT	MATRIXPROD	CALLFUNC
INT128	0.0243	0.0110	0.0151	0.0082
FFT	0.0110	0.0275	0.0086	0.0077
MATRIXPROD	0.0151	0.0086	0.0026	0.0096
CALLFUNC	0.0082	0.0077	0.0096	0.0287

in the original CloudSim (only correcting the MIPS for each stressor), this will give us results as if the multi-core CPUs scales linearly, because that is what is inherently assumed in the original CloudSim implementation.

Of course the number of possible combinations that can be tested is too big, so we will pick up some representative scenarios we think can lead us to conclusions on the effectiveness of the method and the simulators capacity. The first natural scenario is to increase the number of cores, and this will be done for XEON E5-2680v4 only, the reason why is because OPTERON 2214HE and i7-6500U are essentially dual core, remembering that the i7 is quad core only when hyper-threading is enabled, but remembering that hyper-threading was turned off for the testing phase. The tests in Table 12 were performed over all stressors, and represents the overall results we got. Of course some stressors perform better than the others, but remembering that it depends on the multi-core architecture which will be favored, but as the only processor possible was XEON, we prefer to give the consolidated overall data.

Table 12: Testing the model accuracy calculating RMSE between Extended CloudSim simulations and real world XEON E5-2680v4. In addition the RMSE between Extended CloudSim and the Original CloudSim (that uses the linear CPU escalation model). Tests calibrated for 320s of runtime to permit cache stabilization.

	4 cores	6 cores	8 cores	10 cores	12 cores	14 cores
RMSE (Ext.CloudSim vs. XEON)	0.0287	0.0983	0.1356	0.2334	0.4521	0.8034
RMSE (Ext. CloudSim vs. Orig. CloudSim)	0.1498	0.2345	0.2847	0.4334	0.8521	1.5034

Considering the line RMSE (Ext.CloudSim vs. XEON), that represents the model and simulator proposed against the experimental measurements, the RMSE enlarges with respect to the number of cores. With 4 or 6 cores the error is very good, with absolute values always less than 5%. When the number of cores rise up the model show less accuracy, for 8 cores we cannot guarantee a good approximation since up 15% or more individual absolute error showed up. This is an indicator that for a large number of CPUs a higher order behavioral model is going to be needed. Now considering the line RMSE (Orig.CloudSim vs. XEON), that represents the linear scaling model, we see that the RMSE is always worst when compared with the previous line. This means that the model gives us a better prediction on CPU performance scaling than the simple approach of linear scaling model.

5.1.6 REFINING THE MODEL TO IMPROVE RESULTS WITH INCREASED NUMBER OF CORES

Figure 18, shows an test with the number of cores ranging from 4 to 14. The RMSE error grows fast when compared with the measured Xeon performance in the same circunstances (the Ext.CloudSim vs. Xeon plot). In the same Figure we see the Ext. CloudSim vs. Orig. CloudSim , where we see the part of the error, in case of linear performance scaling compared with the BFO model, this shows that the BFO model has always better results since, in this case, the model's predicted performance is less than the linear. The RMSE seems to grow exponentially, and because of it we imagined to compensate this error, since the our model always predict better performance scaling than the measured in experimental systems, when the number of cores is greater than 2. In order to improve the accuracy of the model, we decided to increase the coupling factor by a log factor that depends on the number of cores. Equation

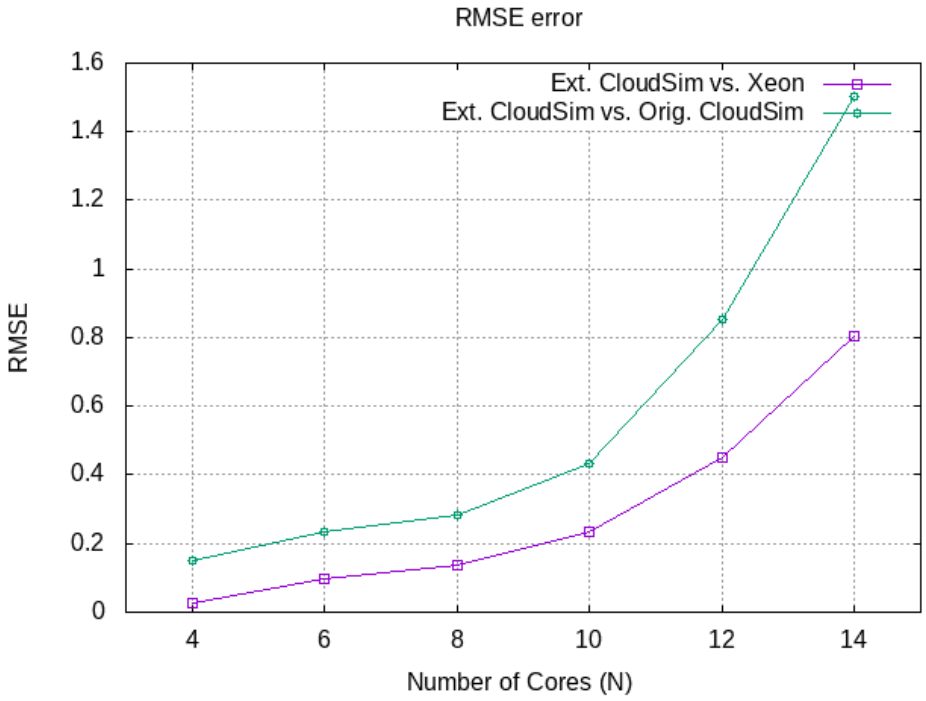


Figure 18: RMSE error vs. Number of cores.

37 shows the proposed correction, where N is the number of cores, being $N \geq 2$, and γ is a non-negative constant to best fit the results, $\gamma \in [0,1]$.

$$\beta' = \beta + \gamma.\beta.\log_2 N = \beta(1 + \gamma.\log_2 N) \ , \tag{37}$$

where β' is the new coupling factor in our extended model.

We performed the same tests over and over with different γ to find if we could reduce substantially the RMSE error. What we found is that, for our log correction, there is a trade off: to make good predictions with N big we sacrifice the error when N is small. However, as the error introduced for small N grow slower than the reduction of the error for N large, we found that $\gamma = 0.1$ can produce good results in the range of our Xeon with 14 cores. Table 13 shows the Extended CloudSim RMSE, repeating first line of Tabel 12, and comparing with the RMSE with Extended CloudSim with γ, in Figure 19 shows these results, where the error still grows with respect to N, but it is much more acceptable, being maximum at 0.1086 at $N = 14$. The "without gamma" line is the same shown in Figure 18 to be compared with our extended model ("with gamma").

Table 13: Testing the improvement in RMSE between Extended CloudSim simulations and real world XEON E5-2680v4 and Extended CloudSim with Correction Factor $\gamma = 0.1$. Tests calibrated for 320s of runtime to permit cache stabilization.

	4 Cores	6 Cores	8 Cores	10 Cores	12 Cores	14 Cores
RMSE (Ext. CloudSim ($\gamma = 0$) vs. XEON)	0.0287	0.0983	0.1356	0.2334	0.4521	0.8034
RMSE (Ext. CloudSim ($\gamma = 0.1$) vs. XEON	0.0293	0.0352	0.0456	0.0614	0.0815	0.1086

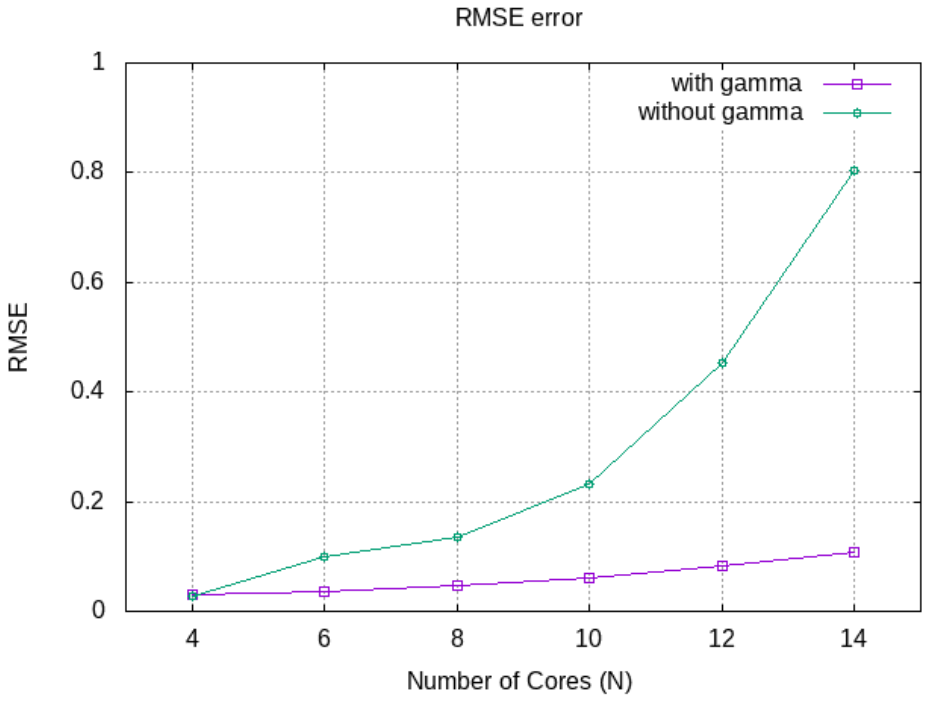


Figure 19: RMSE error vs. Number of Cores. With $\gamma = 0.1$ shows good reduction in RMSE error

6 CONCLUSIONS AND FUTURE RESEARCH

6.1 CONCLUSIONS OF THE MODEL BEHAVIOR AND THE CLOUDSIM EXTENSIONS IMPLEMENTATION

The objective of this study was to produce an extension in the CloudSim tool, by incorporating the BFO model by Kandalintsev and Lo Cigno, in order to improve its performance in the simulation of multicore systems.

The simulator has consistently presented good results in the training phase, with absolute errors of almost zero. Actually this was expected, since the pairwise model is extracted directly from measurements, so we understand the errors are related with the way the simulator operates internally. As the tick of the clock is a fixed value, the simulator always will present an execution time equal or greater than the real, this is caused by a cloudlet could be totally consumed in the middle of a tick, but the completion time is computed in the end of the slice of the time. To improve the quality of the simulation results the tick was set “small enough”, for the tests ran it was fixed in 10 milliseconds. But for really big simulations, where the time of simulation should be optimized we found that a tick of one hundredth of the smallest cloudlet size leads to errors less than 0.2% in average.

During the simulation phase, again the results of the simulator were adherent to the measured during the training phase, and this was expected because the model is a first order model, and the coupling factors are a way to express the scaling behavior using the overhead concept. The RMSE errors are quite small, and should be zero, for the same reasons explained in training phase section.

The generalization part of the simulation presented some interesting results, first one is that the error increases rapidly with the number of CPU cores, for example with 4 cores its is about 2% to 3%, for 6 cores it jumps to 9% and sometimes 10%, and it continues

growing when 14 cores are used, we have error in order of 80%. It suggested that the first order model was not enough to deal with a great number of cores. To improve results of the model we introduced a logarithmic correction that worked well putting the error for 14 cores to about 11%, widening the range where the model could be used. It is something for further investigation if correction law could be used, improving yet more the results. One more remark about γ factor, it is unclear if it is applicable to any pair of applications, since we calculated the RMSE always using one stressor at a time, being the result the arithmetic average among all the tests. It should be clarified in further tests and analysis. Second, the size of the Cloudlets have influence in the accuracy of the simulated results. It is due the fact that, the model used, as well as, the simulator's CPU model consider the system in steady state. However, in real systems, there is transitory responses when the caches are adapting to the demand. For the simulator perspective we can always improve the result by implementing small clock tick, but it elevates the time of a simulation. Users have to have it in mind during the setup.

Additionally, we can list some of the findings and conclusions of this research:

1. First, the implementation of the BFO model in CloudSim has shown good results in all tests and could be a choice for researchers interested in simulating the performance of their own application in multi-core systems. It would demand some time to model its own application behavior. That is not an easy task right now, but we propose an approach to approximate β using application/stressor with similar behavior.

2. Second, in all scenarios of test the implemented model showed better results than the linear escalation assumption, not a single case tested showed the contrary, this rises the confidence in the model, that for the stressors tested was capable to model the performance tendencies. Showing very good adherence with the physical hardware tested up to 6 cores for XEON processors. The results were improved using the log correction using the correction factor $\gamma = 0.1$.

3. Third, the performance counters used to make the measurements in this research, could be the basis to implement on line statistics to calculate the β_s automatically.

4. Fourth, the proposed model is not good for transitory conditions, for example: when the time to stabilize the cache was not negligible the error increased even in two processors tests. It suggests that a transitory model could be studied, in further research, to take into account the cache training time.

5. Fifth, considering the CPU performance scaling laws presented in Chapter 3, the performance of multi-core chips will improve when more software design were implemented to make use of the parallelized computational environment. It tends to be more linear, but what is observed is that very little effort is being done in the software development field to make it a reality. So at least at the moment and probably for some time in the future, the model proposed is still valid and useful.

6.2 LESSONS LEARNED AND FUTURE WORK

The first lesson learned in this research is related to the basic assumption of a β factor could successfully model the CPU performance scaling. This was discovered to be partially true, even if powerful concept, the error accumulates very fast and a correction factor γ had to be introduced, not really solving the problem, but kept the error in a acceptable range.

The second lesson learned is that the BFO model simplified the math of CPU performance scaling, but introduced a very complex situation to deal in the real world, that is the performance needs to be benchmark before being used. It leads to a series of adaptations that pointed to a necessity to deep the analysis of the performance intrinsics, for example the influence of the memory scaling function $g(M)$ of the Memory Bound model could be related to the γ. It is a god point to be studied in a future work.

The third lesson learned is that the extended CloudSim as it is now, in a way to be properly used, depends on a study on how to generate automatically the β_s and γ_s with an online algorithm. Or a study to classify the applications behavior by similar coupling factors β and/or correction factors γ.

The fourth lesson learned is that there is never “free lunch”, a first thought that a simple model can solve all, leaded to a very complex scenario yet fertile in terms of questions yet to be answered.

REFERENCES

AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In: **AFIPS spring joint computer conference**. 1967.

BARROSO, L. A. et al. Rpm: a rapid prototyping engine for multiprocessor systems. **Computer**, v. 28, n. 2, p. 26–34, Feb 1995. ISSN 0018-9162.

BERGAMASCHI, R. A. et al. Performance modeling for early analysis of multi-core systems. In: **In Proceedings of the 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)**. 2007. p. 209–214.

BORKAR, S. Thousand core chipsa technology perspective. In: **2007 44th ACM/IEEE Design Automation Conference**. 2007. p. 746–749. ISSN 0738-100X.

BURGER, D.; GOODMAN, J. R.; KäGI, A. Memory bandwidth limitations of future microprocessors. **SIGARCH Comput. Archit. News**, ACM, New York, NY, USA, v. 24, n. 2, p. 78–89, maio 1996. ISSN 0163-5964. Disponível em: <http://doi.acm.org/10.1145/232974.232983>.

CALHEIROS, R. N. et al. **cloudsim 3.0 API**. Disponível em: <http://www.cloudbus.org/cloudsim/doc/api/index.html>.

CALHEIROS, R. N. et al. Cloudsim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. **Softw. Pract. Exper.**, John Wiley & Sons, Inc., New York, NY, USA, v. 41, n. 1, p. 23–50, jan. 2011. ISSN 0038-0644. Disponível em: <http://dx.doi.org/10.1002/spe.995>.

CALHEIROS, R. N. et al. Cloudsim: A novel framework for modeling and simulation of cloud computing infrastructures and services. **CoRR**, abs/0903.2525, 2009. Disponível em: <http://arxiv.org/abs/0903.2525>.

CARVALHO, C. The gap between processor and memory speeds. In: **In Proceedings of the IEEE International Conference on Control and Automation (ICCA)**. 2002.

CONTE, T. M.; TRACK, E.; DEBENEDICTIS, E. Rebooting computing: New strategies for technology scaling. **Computer**, v. 48, n. 12, p. 10–13, Dec 2015. ISSN 0018-9162.

EYERMAN, S.; EECKHOUT, L. "per-thread cycle accounting in smt processors". In: **Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems**. New York, NY, USA: ACM, 2009. (ASPLOS XIV), p. 133–144. ISBN 978-1-60558-406-5. Disponível em: <http://doi.acm.org/10.1145/1508244.1508260>.

FELDMAN, M.; LAI, K.; ZHANG, l. The proportional-share allocation market for computational resources. v. 20, p. 1075 – 1088, 09 2009.

GROZEV, N. **CloudSim-3.0.3**. 2012. GitHub. Disponível em: <https://github.com/Cloudslab/cloudsim/releases/tag/cloudsim-3.0.3>.

GUÉROUT, T. et al. Energy-aware simulation with dvfs. **Simulation Modelling Practice and Theory**, v. 39, p. 76 – 91, 2013. ISSN 1569-190X. S.I.Energy efficiency in grids and clouds. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1569190X13000786>.

GUSTAFSON, J. Reevaluating amdahl's law. In: . 1988. (Comm. ACM), p. 532–533.

HILL, M. D.; MARTY, M. R. Amdahl's law in the multicore era. **Computer**, v. 41, n. 7, p. 33–38, July 2008. ISSN 0018-9162.

JIANG, J. W. et al. Joint vm placement and routing for data center traffic engineering. p. 2876–2880, 03 2012.

KANDALINTSEV, A. **Application Interference in Multi-Core Architectures: Analysis and Effects**. Tese (phdthesis) — Università degli Studi di Trento, Department of Information Engineering and Computer Science University of Trento, Italy, abr. 2016.

KUNKEL, R. J. E. S. R. et al. A performance methodology for commercial servers. **IBM Journal of Research and Development**, n. 44, p. 851–872, jun. 2000.

LO CIGNO, R.; KANDALINTSEV, A. A behavioral first order cpu performance model for clouds' management. In: **International Congress on Ultra Modern Telecommunications and Control Systems and Workshops**. 2012. p. 40–48. ISBN 978-1-4673-2016-0.

MOORE, G. E. **Cramming more components onto integrated circuits**. April 19 1965.

NANDA, A. K. et al. Memories: A programmable, real-time hardware emulation tool for multiprocessor server design. In: . 2000. v. 35, p. 37–48.

POLLACK, F. J. New microarchitecture challenges in the coming generations of cmos process technologies (keynote address)(abstract only). In: **Proceedings of the 32Nd Annual ACM/IEEE International Symposium on Microarchitecture**. Washington, DC, USA: IEEE Computer Society, 1999. (MICRO 32), p. 2–. ISBN 0-7695-0437-X. Disponível em: <http://dl.acm.org/citation.cfm?id=320080.320082>.

SHRIVASTAVA, V. et al. Application-aware virtual machine migration in data centers. In: **Proceedings - IEEE INFOCOM**. 2011. p. 66 – 70.

SINGH, J. P.; WEBER, W.-D.; GUPTA, A. Splash: Stanford parallel applications for shared-memory. **SIGARCH Comput. Archit. News**, ACM, New York, NY, USA, v. 20, n. 1, p. 5–44, mar. 1992. ISSN 0163-5964. Disponível em: <http://doi.acm.org/10.1145/130823.130824>.

SUN, X.-H.; CHEN, Y. Reevaluating amdahl's law in the multicore era. **J. Parallel Distrib. Comput.**, Academic Press, Inc., Orlando, FL, USA, v. 70, n. 2, p. 183–188, feb 2010. ISSN 0743-7315. Disponível em: <http://dx.doi.org/10.1016/j.jpdc.2009.05.002>.

SUN, X. H.; NI, L. M. Another view on parallel speedup. In: **Proceedings SUPERCOMPUTING '90**. 1990. p. 324–333.

TANG, C. et al. A scalable application placement controller for enterprise data centers. In: **16th International World Wide Web Conference, WWW2007**. 2007. p. 331–340.

TORRELLAS, J.; HENNESSY, J.; WEIL, T. Analysis of critical architectural and programming parameters in a hierarchical. In: **Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems**. New York, NY, USA: ACM, 1990. (SIGMETRICS '90), p. 163–172. ISBN 0-89791-359-0. Disponível em: <http://doi.acm.org/10.1145/98457.98754>.

TRACK, E.; FORBES, N.; STRAWN, G. The end of moore's law. **Computing in Science Engineering**, v. 19, n. 2, p. 4–6, Mar 2017. ISSN 1521-9615.

UBUNTU-WIKI. **stress-ng**. 2008. Disponível em: <https://wiki.ubuntu.com/Kernel/Reference/stress-ng>.

WEAVER, V. M.; DONGARRA, J. Can hardware performance counters produce expected, deterministic results. In: . Atlanta, GA: , 2010.

WIKI, P. **Linux kernel profiling with perf**. 2015. Disponível em: <https://perf.wiki.kernel.org/index.php/Tutorial>.

WULF, W. A.; MCKEE, S. A. Hitting the memory wall: Implications of the obvious. **SIGARCH Comput. Archit. News**, ACM, New York, NY, USA, v. 23, n. 1, p. 20–24, mar. 1995. ISSN 0163-5964. Disponível em: <http://doi.acm.org/10.1145/216585.216588>.

XU, D. et al. Providing fairness on shared-memory multiprocessors via process scheduling. In: **Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems**. New York, NY, USA: ACM, 2012. (SIGMETRICS '12), p. 295–306. ISBN 978-1-4503-1097-0. Disponível em: <http://doi.acm.org/10.1145/2254756.2254792>.

YILMAZ, C. Using hardware performance counters for fault localization. v. 0, p. 87–92, August 2010.

ZAPARANUKS, D.; JOVIC, M.; HAUSWIRTH, M. Accuracy of performance counter measurements. In: . Boston, Massachusetts, USA: , 2009. (in Proc. of IEEE ISPASS), p. 23–32. Disponível em: <https://www.researchgate.net/publication/258340962_Accuracy_of_Performance_Counter_Measurements>