Tendências em Matemática Aplicada e Computacional, **20**, N. 1 (2019), 169-188
© 2019 Sociedade Brasileira de Matemática Aplicada e Computacional
www.scielo.br/tema
doi: 10.5540/tema.2019.020.01.0169

Optimization of the Size of Thread Pool in Runtime Systems to Enterprise Application Integration: A Mathematical Modelling Approach

D.L. FREIRE, R.Z. FRANTZ,
F. ROOS-FRANTZ and S. SAWICKI

Received on February 28, 2018 / Accepted on December 07, 2018

ABSTRACT. Companies seek technological alternatives that provide competitiveness for their business processes. One of them is integration platforms, software tools that build integration solutions, which allow the different applications that make up the software ecosystem to work synchronously and that new applications or functionality be incorporated with the least impact in the existing ones. The runtime system is the component of the integration platform responsible for managing the computational resources that run the integration solution. Among these resources are the processing units, called threads, or sets of those threads, called thread pools. The performance of the runtime systems is directly related to the number of threads available to run the integration solution, but scaling the number of threads that provide a shorter response time is a challenge for software engineers. If this quantity is undersized, it may cause a delay in the execution; if it is overestimated, it could cause a waste of computational resources. This article presents a mathematical model, defined by differential equations, that establishes the optimum number of threads, which maximizes the expected performance gain by minimizing the execution time of the integration solution. In addition, it presents the mathematical model application, which assists the analysis of the expected gain in different architecture scenarios and quantity of threads.

Keywords: enterprise application integration; multithread programming; runtime system; mathematical modelling; integration platforms.

1 INTRODUCTION

The set of applications that compose the software ecosystem of companies is usually heterogeneous, because such applications have been acquired over time, without the concern of communication between them [15]. In addition, technological advancement has led to the incorporation of cloud computing software services, what has left software ecosystems even more heterogeneous.

*Corresponding author: Daniela L. Freire – E-mail: dsellaro@unijui.edu.br – `http://orcid.org/0000-0002-5363-3608`
Universidade Unijuí, Departmento de Ciências Exatas e Engenharia, Ijuí-RS, Brazil. E-mails: dsellaro@unijui.edu.br, rzfrantz@unijui.edu.br, frfrantz@unijui.edu.br, sawicki@unijui.edu.br

The business processes of the company need to provide fast and reliable answers, and for that, their applications and services need to be prepared to work together. Enterprise Application Integration (EAI) is the research field that provides methodologies, techniques and tools to support the development of integration solutions, which allow the different applications of the ecosystem to work synchronously and enable new applications to be incorporated, with the least impact on the existing ones [17].

With the growth of multi-core architectures, the industry has started to focus on multithreaded applications [41]. Thereby, several tools have been released to improve the performance of the applications [22, 47, 23, 7]. Considering this scenario, some integration platforms that support threads creation and management have been developed, such as Mule [10], Camel [20], Spring Integration [13], Petals [42], WSO2 ESB [21] and Guaraná [14, 16].

Integration platforms are specialized software tools to design, implement, monitor and execute integration solutions. An integration solution implements a workflow composed of different atomic tasks that run through that flow [18]. Typically, these platforms provide a domain-specific language, development toolkit, monitoring tools, and a runtime system. The specific-domain language allows the creation of conceptual models for the integration solution. The development toolkit is a set of tools that allows transforming the conceptual model into an executable code. Monitoring tools are used to detect failures that may occur during the execution of an integration solution. The runtime system provides the support needed to run the integration solution [19].

The tasks that make up an integration solution are executed by computational resources present in the runtime system, in this article called execution threads. Threads are used in multithread programming to permit simultaneous execution of tasks. Multithread programming architecture follows two models: thread-per-request and threads pool [37]. The former generates a thread for each execution request of a task, and it is destroyed at the end of the execution. In threads pool, threads are created and maintained; the latter executes the task and, at the end of the execution, it releases the thread to the pool.

In the thread pool model, it is possible to set parameters, such as: number of threads in the pool, maximum number of threads allowed in a pool, maximum time interval that a thread will be idle waiting for a new task, among others. Experimental studies suggest that thread pool architectures can have an impact on performance [1, 37, 25] and, to keep computing resources working properly, the supply of these resources must accompany variations in the workload requested [8].

Experimental researches indicate that a thread pool model can significantly improve system performance and also reduce response time [49, 34, 37]. These benefits make thread pool systems to be adopted by a large number of popular applications. Despite the advantages, multithread programming is more complex because it meets both quality attributes and performance parameters. The configuration of the threads depends on the empirical knowledge of software engineers, who must scale the number of threads in the pools in the runtime systems, in order to provide adequate performance, achieving a shorter response time and a greater workload in order to fulfill the requisitions. If the number of threads in the pool is greater than necessary, it can cause a waste of computing resources, since at a low demand moment the threads will be idle. In contrast, if the number of threads in the pool is lesser than required, it will cause a slower execution and may fail to meet quality attributes [6, 4].

It is possible to consider that the performance of the applications also depends on the throughput provided by the thread pool. In this case, an important aspect to determine the performance of the thread pool is the size of the pool. The larger the thread pool, the more simultaneous tasks can be handled, and a faster response time is provided. However, when the thread pool size increases, the management overhead can degrade system performance. In this way, dealing with this trade-off is important for the system optimization [49].

This article proposes a mathematical formulation that characterizes the costs associated with adopting the thread-by-request and thread pool architectures and obtains the optimum size of the thread pool, maximizing the expected gain by minimizing the execution time of a solution. Furthermore, the article presents the application of the mathematical formulation, comparing the expected gain to the use of the thread pool architecture of the related different costs of using the threads.

The remainder of this article is organized as follows. Section 2 discusses works related to threads optimization. Section 3 develops the mathematical formulation of the problem. Section 4 brings the application of the mathematical formulation. Finally, Section 5 presents the conclusions.

2 RELATED WORK

Many different types of techniques for dynamically optimizing the number of threads have been proposed in the literature. Some approaches consider schemes to predict the optimal thread pool [49, 30, 43, 29], other works use dynamic feedback and runtime information [31, 2, 32]. There are also researches aimed at studying power performance trade-offs [9, 28, 39, 3, 32, 38], effect of OS level factors [12, 11, 46, 48, 33] and scheduling techniques [26, 2, 45].

Lee et al. [27] present a dynamic system that automatically adjusts the number of threads in an application in order to optimize system efficiency. Using a dynamic compilation system, the authors developed an application called *Thread Tailor*, which combines threads by communication patterns to decrease synchronization overhead. Thread Tailor uses off-line analysis to predict what type of threads will exist at runtime and the communication patterns between them based on the architecture, dynamic system state, and communication and synchronization relationships between threads. They used a baseline of number of threads equal to the number of cores for performance comparisons and dynamically leverage the code generation to optimize away unnecessary synchronization after combining threads. Moreover, they emphasize that there is significant limitation of most OS-based thread scheduling researches. According to the authors, these techniques typically do not recognize communication and synchronization patterns that provide important hints as to where and when threads should leverage. Wu et al. [48] observed that hardware or OS techniques may not be able to infer enough information from the application code, aiming at the most efficient adaptation points. In this way, to determine the number of threads required to execute an application, probably the OS and hardware cannot infer enough information concerning the application to make efficient choices, indicating, with this, the use of dynamic compilation.

Suleman et al. [41] propose a framework to dynamically control the number of threads at runtime based on the application behaviour. They use a simple analytical model that captures the impact of data synchronization at execution time. This technique checks a piece of an application parallel region and executes it sequentially to find synchronization and communication elements between them. Afterwards, it analyses these points to estimate the optimal number of threads for each region. In this work, the authors demonstrate that there is no advantage of using a number of threads larger than the number of cores. In this sense, other researchers have made similar discussions. Nieplocha et al. [32] demonstrated on real hardware that some applications saturate shared resources in the Sun Niagara processor with only 8 threads, although the hardware has support for 32 simultaneous threads. Saini et al. [35] made similar observations with different types of processors about performance degradation. In contrast, Pusukuri et al. [33] demonstrate that, in a 24-core system, many of the PARSEC programs require much more than 24 threads to maximize speedups.

Jung et al. [24] presented performance estimation models and techniques for generating an adaptive execution code for simultaneous multithreading (SMT) architectures. The adaptive execution techniques determine an optimal number of threads by means of dynamic feedback and time of execution. With this, a compiler preprocessor generates a code that automatically determines at runtime the optimal number of threads for each parallel loop in the application. The authors avoid executing some parallel loops in parallel or change the number of threads to run the loops optimally if the performance degradation of the loops exceeds a predefined threshold value at runtime. Both articles, by Jung and Suleman, propose monitoring the execution and using similar models for predicting the appropriate number of threads of a given system state. Agrawal et al. [2] present also an adaptive task scheduler technique that provides continual parallel feedback for the job scheduler of the application. Similarly, Pusukuri et al. [33] presented a technique for dynamically determining the appropriate number of threads without recompiling the application or using complex compilation techniques or modifying Operating System policies. They developed a framework called *Thread Reinforcer*. According to the authors, not only is the Thread Reinforcer effective in selecting the number of threads, it also has very low runtime overhead. Other researches that consider scheduling techniques based on different application characteristics and dynamic estimates of the system resources usage were also presented by [51, 45] and [26]. These approaches, however, only allocate threads when necessary, and do not consider the impact of the number of threads in the applications. This behaviour was discussed by [27], who observed that the OS and hardware cannot infer enough information concerning the application to make efficient choices as to determining the number of threads than an application should leverage.

Schwarzrock et al. [38] state that parallel applications are usually executed using the maximum number of threads allowed by the hardware available to maximize performance. They consider that this approach may not be the best when it comes to energy efficiency and may even lead to performance reduction in some particular cases. The authors demonstrated that there is a potential gain by automatically adapting the number of threads during runtime in the multithread application OpenMP considering the trade-off between energy save and performance. Lorenzon et al. [31] also investigated general-purpose and embedded multicores to achieve optimal trade-offs between performance and energy. Similarly, [5, 39, 3, 9] and [28] can also dynamically adjust the number of threads based both on performance and power optimization.

Some researchers have proposed schemes to predict the optimal thread pool size based on heuristic elements [30]. However, this strategy usually is hard due to complexity and overhead. In contrast, Xu et al. [49] developed a set of performance metrics for quantitatively analysing the thread pool performance. Similarly, a methodology to identify threads with performance deviations in pools based on the dissimilarity of their resource usage metrics was proposed by [43]. The work presented by [29] proposes a dynamic thread pool method to solve the high concurrency problems. According to the authors, the dynamic thread pool method is more efficient than the traditional pool.

Our approach, however, characterizes the costs associated with adopting the thread-per-request and thread pool architecture and obtains the optimum size of the thread pool, maximizing the expected gain and minimizing the execution time of a solution. For that, it presents the application of the mathematical formulation, comparing the expected gain with the use of the thread pool architecture of the related different costs of using the threads.

3 PROBLEM FORMULATION

This section generically describes the operation of the current model of runtime systems of integration platforms and introduces a mathematical formulation to determine the optimum number of threads for their pools and the gain obtained with this configuration of the thread pool.

3.1 Current model

The implementation of concurrency or parallelism in the execution of tasks is complex in most programming languages[50]. Originally, the mechanism for concurrent programming of programming languages was inspired by the concurrency principles of operating systems. Therefore, languages have been incorporating new features into this mechanism, such as the use of threads to execute program algorithms in the so-called multithread programming.

With the concurrent execution of software tasks, when some threads are locked, waiting for some operation, others may be performing tasks. Threads can share a single address space and all their data, as well as be easily created and destroyed because they do not have any resources associated with them. This fact is very useful when the number of required threads changes dynamically and quickly. In addition, the use of threads provides a performance gain in applications that have high computational effort and I/O, since they allow these activities to overlap[44].

The classes in the Java API provide settings, which give flexibility to thread pools, such as: indicate the number of threads kept in the pool, even without executing; indicate the maximum number of threads allowed in a pool, that is, how much a thread pool can grow; determine the maximum amount of time that a surplus thread will idle, waiting for a new task; define the type of queue used to hold the tasks before execution [36]. Figure 1 illustrates a task queue and a thread pool of a runtime system of an integration platform.

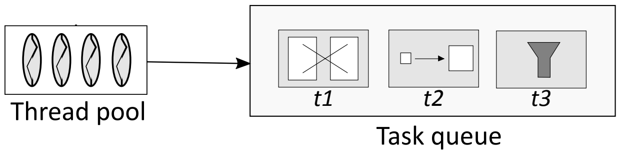

Figure 1: Thread pool and task queue of a runtime system.

There are several task queues options in the Java *concurrent API* which serve different needs such as: messaging, producer-consumer, parallel tasks. Some of them work as a limited *buffer*, where the tasks are kept as elements of an array and the tasks are selected by the *First-In-First-Out (FIFO)* policy; others work with an unlimited queue and organized in a specified order of priority, where at the beginning of the queue is the task with the highest priority; others allow the tasks to be scheduled with a predefined delay time for their execution and can only be obtained from that queue when that delay time expires. At the beginning of the queue is the task that has expired the longest.

The task queue interacts with the size of the pool as follows: *(i)* if there are fewer threads running in the pool than the amount that has been set, a new thread will always be added to the pool instead of queuing more tasks ; *(ii)* if the number of threads running in the pool is equal to or greater than the amount that was set, more tasks will be queued rather than adding a new thread to the pool; *(iii)* if a task request cannot be queued, a new thread is created as long as the number of threads running is smaller than the maximum limit, otherwise, the task will be rejected. The runtime system of an integration platform is a mechanism capable of providing hardware and software capabilities that allow defining how an integration solution should be performed, regardless of the programming language that is being used by the integration solution. In a more comprehensive definition, it can be said that the runtime system is the engine that dynamically determines the behaviours that an integration solution will have during execution.

The most common runtime system models have the following main elements: scheduler, task queue, task, threads, monitors. The scheduler is the central element of the runtime system. It manages all the runtime system activities. The scheduler has a task queue, a set of threads, and monitors. The task queue stores the tasks in the order of priority in which they should be processed. Each task can only be executed when it reaches the execution time for which it was scheduled. The default is that this execution time is a unit of time after its entry in the queue, that is, that the task is immediately ready to be executed. If the task has been scheduled for a future time, this task will have to wait for this deadline to expire. Each available execution thread checks the task queue and performs the task according to its schedule. The available threads process the tasks concurrently.

The monitors provide statistics on memory usage, CPU and on the task queue, recording information such as the percentage of memory usage, system time, usage time consumed for execution, queue size, and total number of tasks that were processed. Monitors are run by specific threads that become active at regular intervals of time units defined by software engineers to record the information and store it in a file and then become inactive until the next interval. The settings for the scheduler are usually made through an XML file, which contains: number of threads, files for statistics generated by monitors, monitors running frequency and logging system to notify about warnings and errors. Scheduler initialization takes place by loading and parsing the configuration file, initializing the logging system, and creating the job queue. The runtime system is not initialized when it is created, but when the software engineer decides to start it. When the scheduler is started, the monitors and threads are started too. Threads are enabled to recurrently check the task queue for ready-to-run tasks. This strategy causes threads to keep running tasks, as long as there are tasks to be performed.

3.2 Mathematical analysis

Below is presented a mathematical analysis to determine a model for the runtime gain with the use of a thread pool and an optimal pool size to maximize this gain. For this analysis, the following premises are assumed:

- each thread pool has the same execution priority and receives an equal portion of CPU time;
- the performed tasks have similar computational complexity, and there is no significant difference in relation to the use of CPU and memory required to execute them;
- the overall costs of processing the pool are calculated from the weighted sum of the factors that affect the processing latency, so it does not consider each specific factor that impacts the time elapsed between the execution request of the task until the completion of its execution.

Two costs are considered in units of time: the first, c_1, is the cost associated with creating and destroying a thread individually; the second, c_2, is the cost of maintaining and executing threads in a pool. Assuming that the same operating system and same integration solution is used, c_1 and c_2 are constants. Also, the cost of creating and destroying a thread that is not bound to a pool is greater than the cost of assigning and releasing threads in a thread pool, that is, $c_1 \gg c_2$ [30].

The cost associated with a pool of size n when the total number of concurrently running threads is x is shown in Table 1. It compares the costs of not having a pool, adopting the thread-by-request architecture, with the costs of the pool, adopting the thread pool architecture, obtaining the gain of a pool of size n by the difference between these cases.

Table 1: Comparison between the threads-by-request architecture and the thread pool architecture.

Case	Costs of the architecture		Expected gain
	threads-per-request	threads pool	
$1 \leq x \leq n$	$c_1 . x$	$c_{2.}\ n$	$(c_{1.}\ x - c_2 . n)$
$x > n$	$c_1 . x$	$c_{2.}\ n + c_1 . (x - n)$	$n . (c_1 - c_2)$

- $0 < x \leq n$: the number of threads running is lesser than the number of threads available in the pool. In this case, the pool is sufficient to execute the task demand and the cost will be $c_2 \cdot n$, this is the cost of maintaining the n threads in the pool. With the adoption of the threads-per-request architecture, the cost will be $c_1 \cdot x$, this is the cost, c_1, of creating and destroying the number of threads being executed, represented by the variable x. Therefore, the gain in adopting the pool architecture is the difference between $c_1 \cdot x$ and $c_2 \cdot n$.
- $x > n$: the number of tasks running is greater than the number of threads in the pool, so the cost in the thread pool architecture will be increased by the cost of creating additional threads to supply the demand, $c_1 \cdot (x - n)$, resulting in an equal total cost to $c_2 \cdot n + c_1 \cdot (x - n)$. Then, in the second case, the gain in adopting the threads pool architecture is given by the difference between $c_1 \cdot x$ and $c_2 \cdot n + c_1 \cdot (x - n)$.

The number of threads running simultaneously tends to vary throughout the execution time of an integration solution, depending on some factors such as the computational complexity of the tasks being performed, message size and message input rate. The probabilistic behaviour of a random variable will be described by its probability density function. A probability density function is a function p(x) that satisfies the following properties:

- $p(x) \geq 0$
- $\int_{-\infty}^{\infty} p(x)\mathrm{d}x$
- Given a function $p(x)$ satisfying the above properties, then $f(x)$ represents some continuous random variable X, so that:

$$P(a \leq X \leq b) = \int_a^b p(x)\mathrm{d}x$$

The probabilities associated with a continuous random variable X can be calculated from the distribution function. Given a random variable X, the distribution function of X is defined by:

$$F_X(x) = P(X \leq X) \forall x \in \mathbb{R}$$

By the *Fundamental Theorem of Calculus*, the probability density function is the derivative of the distribution function:

$$p(x) = \frac{\mathrm{d}}{\mathrm{d}x} F_X(x)$$

By definition, if X is a continuous random variable and $h : \mathbb{R} \to \mathbb{R}$ is any function, then $Y = h(X)$ is a random variable and its expected value E is given by:

$$E[h(X)] = \int_{-\infty}^{\infty} h(x) \cdot p(x)\mathrm{d}x$$

To compute the gain of adopting the thread pool architecture, it is assumed that this quantity represented is a random variable with the probability distribution $f(x)$. Equation 3.1 expresses the expected gain, $E(n)$, in adopting the thread pool architecture [40]:

$$E(n) \quad = \quad \sum_{r=o}^{n} (c_1 \cdot r - c_2 \cdot n).f(r) \quad + \quad \sum_{r=n+1}^{\infty} (c_1 \cdot n - c_2 \cdot n).f(r) \tag{3.1}$$

Getting the optimal pool size is equivalent to finding the number of threads for the pool, n^*, which generates the highest gain, which means minimizing the associated costs. This gain, $E(n^*)$, can be expressed as in Equation 3.2:

$$E(n^*) = \sup E(n) \colon n \in N \tag{3.2}$$

The discrete probability $f(x)$ can be replaced by $p(x) \cdot \mathrm{d}x$, where $p(x)$ is the probability density, leaving the expected gain for the pool expressed as in Equation 3.3:

$$E(n) \quad = \quad \int_0^n (c_1 \cdot r - c_2 \cdot n) \quad \cdot \quad p(r) \cdot \mathrm{d}r \quad + \quad \int_n^{\infty} (c_1 \cdot n - c_2 \cdot n) \quad \cdot \quad p(r) \cdot \mathrm{d}r \tag{3.3}$$

The optimal pool size that maximizes the expected gain by adopting the thread pool architecture can be found by the first order derivative, shown in Equation 3.4:

$$\frac{\partial E}{\partial n} = -c_2 + c_1 \cdot \int_{n^*}^{\infty} p(x) \cdot \mathrm{d}x = 0 \tag{3.4}$$

Deriving Equation 3.4, the second order derivative is obtained in relation to n and the expected gain is kept positive if the condition is satisfied:

$$\frac{\partial^2 E}{\partial n^2} = -c_1 \cdot p(x) \leq 0 \tag{3.5}$$

$\zeta = c_2/c_1$ is defined as the cost ratio of keeping a thread in the pool, c_2, for the cost of creating and destroying a thread c_1, and dividing Equation 3.4 for c_1:

$$\int_{n^*}^{\infty} p(x) \cdot \mathrm{d}x = c_2/c_1 = \zeta. \tag{3.6}$$

Then,

$$\int_o^{\infty} p(x)\cdot \mathrm{d}x = \int_o^{n^*} p(x)\cdot \mathrm{d}x + \int_{n^*}^{\infty} p(x)\cdot \mathrm{d}x = 1$$

it results,

$$\int_o^{n^*} p(x)\cdot \mathrm{d}x = 1 - c_2/c_1 = 1 - \zeta. \tag{3.7}$$

Since pool size is an integer, it can be determined by Equation 3.8, where $\lfloor s \rfloor$ is the next integer smaller than s.

$$\begin{aligned} \int_o^{\lfloor n^* \rfloor} p(x)\cdot \mathrm{d}x \leq 1 - c_1/c_2 \therefore \int_o^{\lfloor n^* \rfloor} p(x)\cdot \mathrm{d}x \leq 1 - \zeta \\ \int_o^{\lfloor n^*+1 \rfloor} p(x)\cdot \mathrm{d}x > 1 - c_1/c_2 \therefore \int_o^{\lfloor n^*+1 \rfloor} p(x)\cdot \mathrm{d}x > 1 - \zeta \end{aligned} \tag{3.8}$$

By Equation 3.8, the optimal size of the pool is proportional to ζ. The higher the cost of creating threads, c_1, or the lower the maintenance cost of the pool, c_2, the larger the pool size will be.

Equation 3.7 and Equation 3.8 show that the ideal thread pool size, n^*, depends not only on ζ, but also the workload of the integration solution, which is the number of tasks being performed, represented by the probability density, $p(x)$. Equation 3.8 considers these associated costs and obtains optimal pool size by maximizing the expected gain.

4 EXPERIMENTAL RESULTS

This section shows the application of the mathematical formulation to find the optimal size of the thread pool, considering the cost associated with creating and destroying a thread individually, c_1, and the cost of maintaining and executing threads from a pool, c_2, considering different probability density functions $p(x)$. Additionally, we present graphics show that the gain of a pool of size n corresponds to the results found analytically with the proposed mathematical model. The expected gain is defined as the difference between the associated costs in the adoption of the thread pool and in the adoption of the thread-per-request architecture.

4.1 Research question

To achieve the goal of this work, we seek to answer the following research question:

Is it possible to provide a mathematical formulation to obtain the optimum size of the thread pool of a runtime systems of integration platforms, maximizing the expected gain by minimizing the execution time of an integration solution?

To answer this question, we have proposed a mathematical model, defined by differential equations, which was presented in Section 3.2. With this research, we are aiming to contribute to a novel approach focused on Enterprise Integration Application area to deal with thread pool size.

4.2 Variables

By means of the mathematical models represented by Equation 3.3 and by Equation 3.8 are obtained the optimal thread pool size of runtime systems of integration platforms and the expected gain with the use of this number of thread in the pool in terms of saving time in the adoption of the thread pool architecture. Therefore, the measured variables are:

- n^* - the optimal number of threads in the pool.
- $E(n^*)$ - expected gain.

The optimal size of the pool depends of the cost of creating threads c_1, on the cost of maintenance of the pool c_2, and on the workload of the integration solution, which is the number of tasks being performed, represented by $p(x)$.

4.3 Scenarios

In this application of the mathematical formulation, forty scenarios are considered in order to obtain the optimal number of threads in the pool, using four different probability density functions and ten different cost ratios of keeping a thread in the pool, and sixteen scenarios in order to obtain the expected gain as a function of the number of pool threads, using four different probability density functions, four different cost ratios and the number of threads in the pool varying continuously in a given range.

In order to obtain the optimal number of threads in the pool n^*, four probability density functions with ten values of cost ratio are used. The set of values that ζ can assume is represented by $\mathbb{Z}$, where $\mathbb{Z} = \{$ 0.000001, 0.00001, 0.0001, 0.001, 0.005, 0.01, 0.1, 0.5, 0.8, 1$\}$.

The scenarios are:

- $p(x)$ represented by a uniform distribution with $\zeta \in \mathbb{Z}$.
- $p(x)$ represented by an exponential density with $\zeta \in \mathbb{Z}$.
- $p(x)$ represented by a density of Pareto with $\zeta \in \mathbb{Z}$
- $p(x)$ represented by a Gama density with $\zeta \in \mathbb{Z}$.

Aiming to achieve the expected gain as a function of the number of threads in the poo $E(n)$, four probability density function with four values of cost ratio and number of threads varying continuously from 0.8 to 1 are used.

The scenarios are:

- $p(x)$ equal to a uniform distribution using $\zeta \in \{0.01, 0.1, 0.5, 0.8\}$.
- $p(x)$ equal to an exponential density using $\zeta \in \{0.01, 0.1, 0.15, 0.2\}$.

- $p(x)$ equal to a density of Pareto using $\zeta \in \{0.1, 0.15, 0.2, 0.5\}$.
- $p(x)$ equal to a Gama density using $\zeta \in \{0.1, 0.15, 0.2, 0.5\}$.

4.4 Execution

In the cases listed below, the optimal size of a thread pool is determined from the equations 3.7 and 3.8 and it is assumed that probability density is:

1. a uniform distribution,
$$p(x) = \begin{cases} 0.1, & \text{if} \quad 0 \leq x \leq 10, \\ 0, & \text{otherwise.} \end{cases} \tag{4.1}$$
Then,
$$\int_o^{\lfloor n^* \rfloor} 0.1 \cdot \mathrm{d}x \leq 1 - \zeta \therefore \lfloor n^* \rfloor \cdot 0.1 \leq 1 - \zeta$$
$$\int_o^{\lfloor n^*+1 \rfloor} 0.1 \cdot \mathrm{d}x > 1 - \zeta \therefore \lfloor n^* + 1 \rfloor \cdot 0.1 > 1 - \zeta$$
The optimal size of the pool is $n^* = 10 \cdot (1 - \zeta)$.
Assuming $\zeta = 0.1$ seconds, we have the optimal pool size $n^* = 9$.

2. an exponential density,
$$p(x) = e^{-x} \tag{4.2}$$
Then,
$$\int_o^{\lfloor n^* \rfloor} (e^{-x}) \cdot \mathrm{d}x \leq 1 - \zeta \therefore -e^{\lfloor -n^* \rfloor} + 1 \leq 1 - \zeta$$
$$\int_o^{\lfloor n^*+1 \rfloor} (e^{-x}) \cdot \mathrm{d}x > 1 - \zeta \therefore -e^{\lfloor -(n^*+1) \rfloor} + 1 > 1 - \zeta$$
The optimal size of the pool is: $n^* = -\ln(\zeta)$.
Assuming $\zeta = 0.01$ seconds, we have the optimal pool size $n^* = 5$.

3. a density of Pareto,
$$p(x) = \begin{cases} \frac{1}{x^2}, & \text{if} \quad 0 \leq x \leq 1, \\ 0, & \text{otherwise.} \end{cases} \tag{4.3}$$
Then,
$$\int_o^{\lfloor n^* \rfloor} \frac{1}{x^2} \cdot \mathrm{d}x \leq 1 - \zeta \therefore 1 - \frac{1}{\lfloor n^* \rfloor} \leq 1 - \zeta$$
$$\int_o^{\lfloor n^*+1 \rfloor} \frac{1}{x^2} \cdot \mathrm{d}x > 1 - \zeta \therefore 1 - \frac{1}{\lfloor n^* + 1 \rfloor} \cdot 0.1 > 1 - \zeta$$
The optimal size of the pool is $n^* = \frac{1}{\zeta}$.
Assuming $\zeta = 0.1$ seconds, we have the optimal pool size $n^* = 10$.

4. a Gama density,

$$p(x) = \begin{cases} x \cdot e^{-x}, & \text{if} \quad x > 0, \\ 0, & \text{otherwise.} \end{cases} \tag{4.4}$$

Then,

$$\int_o^{\lfloor n^* \rfloor} x \cdot e^{-x} \mathrm{d}x \cdot \leq 1 - \zeta \therefore 1 - (\lfloor n^* \rfloor + 1) \cdot e^{\lfloor -n^* \rfloor} \leq 1 - \zeta$$
$$\int_o^{\lfloor n^*+1 \rfloor} x \cdot e^{-x} \cdot \mathrm{d}x > 1 - \zeta \therefore 1 - (\lfloor n^* + 1 \rfloor + 1) \cdot e^{\lfloor -(n^*+1) \rfloor} > 1 - \zeta$$

The optimal size of the pool is: $(n^* + 1) \cdot e^{-n^*} = \zeta$.

Assuming $\zeta = 0.09$ seconds, we have the optimal pool size $n^* = 4$.

Equation 3.3 is used to calculate the expected gain as a function of the number of pool threads. In the first case, considering the probability density by the uniform distribution provided by Equation 4.1, Equation 3.3 results in:

$$E(n) = -0.05 \cdot c_1 \cdot n^2 + (c_1 - c_2) \cdot n \tag{4.5}$$

In the second case, considering the density of probability by the exponential function provided by Equation 4.2, Equation 3.3 results in:

$$E(n) = c_1 \cdot (1 - e^{-x}) - c_2 \cdot n \tag{4.6}$$

In the third case, considering the density of probability by the density of Pareto provided by Equation 4.3, Equation 3.3 results in:

$$E(n) = c_1 \cdot (1 + \ln n) - c_2 \cdot n \tag{4.7}$$

In the forth case, considering the density of probability by the Gama density provided by Equation 4.4, Equation 3.3 results in:

$$E(n) = c_1 \cdot (2 - (n+2) \cdot e^{-n}) - c_2 \cdot n \tag{4.8}$$

4.5 Results and Discussion

The results found for the optimal number of threads in a pool using ten values of ζ are shown in Table 2.

In the first case, with probability density being a uniform distribution, it is found that the optimal pool size, n^*, is 10 for ζ ranging from 0.000001 to 0.01 seconds; 9 for ζ equal to 0.1; 5 for ζ equal to 0.5; 2 for ζ equal to 0.8; and 1 for ζ equal to 1.

Table 2: Optimum pool size versus cost.

ζ	n^* uniform distribution	exponential density	density of Pareto	density Gama
0.000001	10	14	10^6	17
0.00001	10	12	10^5	14
0.0001	10	9	10^4	12
0.001	10	7	10^3	9
0.005	10	5	200	7
0.01	10	5	100	6
0.1	9	2	10	4
0.5	5	1	2	2
0.8	2	1	1	1
1	1	1	1	1

In the second case, with probability density being an exponential, n^* is 14 for ζ equal to 0.000001; 12 for ζ equal to 0.00001; 9 for ζ equal to 0.0001; 7 for ζ equal to 0.001; 5 to ζ ranging from 0.005 to 0.01; 2 for ζ equal to 0.1; 1 for ζ ranging from 0.5 to 1.

In the third case, with probability density of Pareto, n^* is 10^6 for ζ equal to 0.000001; n^* is 10^5 for ζ equal to 0.00001; 10^4 for ζ equal to 0.0001; 10^3 for ζ equal to 0.001; 200 to ζ equal to 0.005; 100 to ζ equal to 0.01; 10 for ζ equal to 0.1; 2 for ζ equal to 0.5; and 1 for ζ ranging from 0.8 to 1.

In the forth case, with probability density Gama, n^* is 17 for ζ equal to 0.000001; n^* is 14 for ζ equal to 0.00001; 12 for ζ equal to 0.0001; 9 for ζ equal to 0.001; 7 to ζ equal to 0.005; 6 to ζ equal to 0.001; 4 for ζ equal to 0.1; 2 for ζ equal to 0.5; and 1 for ζ ranging from 0.8 to 1.

The variation of the expected gain regarding the number of threads in the pool with different values of ζ, where $\zeta = c2/c1$ in seconds, when the probability density is a uniform distribution, an exponential, a density of Pareto or a Gama density; according to the equation 4.5, 4.6, 4.7 and 4.8, respectively, is shown in Figure 2. The larger black dot on the curves shows the number of threads that provides the highest expected gain for each of the probability density functions.

With probability density being a uniform distribution, n^* is 10 for ζ equal to 0.01; 9 for ζ equal to 0.1; 5 for ζ equal to 0.5; 2 for ζ equal to 0.8. With probability density being an exponential, it is found that the optimal pool size, n^*, is 5 for ζ equal to 0.01 seconds; 2 for ζ equal to 0.1, 0.15 or 0.2 seconds. With probability density of Pareto, n^* is 10 for ζ equal to 0.1; n^* is 7 for ζ equal to 0.15; 5 for ζ equal to 0.2; 2 for ζ equal to 0.5. With probability density Gama, n^* is 4 for ζ equal to 0.1; n^* is 3 for ζ equal to 0.15 or 0.2; 2 for ζ equal to 0.5.

In all cases studied, the total gain increases with the size of the thread pool until it reaches n^*, afterwards, it begins to decrease, confirming that adding threads to the pool from that value on

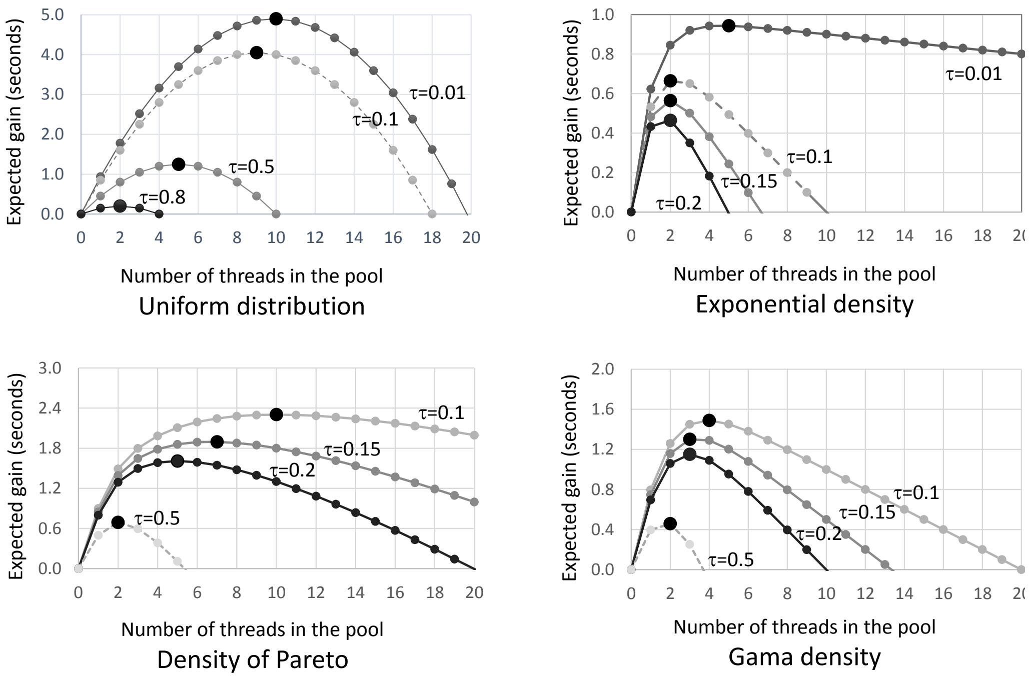

Figure 2: Expected gain versus number of threads.

does not improve the expected gain. In addition, the expected gain of a thread pool is inversely proportional to the cost relation. It is found that the gain is lower, when $\zeta = c_2/c_1$ is high, since the thread maintenance cost is increased.

5 CONCLUSION

To keep up with technological trends and optimize the results of their business processes, companies are seeking to integrate the applications of their software ecosystem. Integration platforms are software tools that support the construction of integration solutions, which interconnect the applications, making them work synchronously. The runtime system is the component of the integration platforms responsible for running the integration solutions and must offer adequate performance and efficient use of computing resources so that the solution produces faster results without financially burdening companies.

One of the challenges faced by software engineers is to empirically scale the number of threads in the thread pool contained in the runtime systems. When the number of threads in the pool is oversized, it causes a waste of computational resources and, consequently, a financial waste; when undersized, it leads to a slower execution, which may fail to meet the quality attributes. This article proposed a mathematical formulation for the expected gain according to the number of threads, considering the costs of maintaining the thread pools in relation to the costs of creating threads as the workload of the integration solution requires. By means of this formulation, we obtained the optimum size of the thread pool, which maximizes the expected gain by minimizing the execution time of the solution.

To verify the mathematical formulation, the expected gain was calculated with four different functions to represent the probability distribution of the number of threads simultaneously running in an integration solution and with different cost values related to the two thread utilization options. Applying the mathematical formulation, it was verified that the total gain increases with the increase of the number of threads in the pool to the optimum size, and then this gain begins to decrease. In addition, it was also found that the expected gain of a thread pool is inversely proportional to the ratio between the cost of maintaining the thread pool and the cost of creating threads individually on demand.

ACKNOWLEDGEMENTS

This work was supported by the Brazilian Co-ordination Board for the Improvement of University Personnel (CAPES) and by the Research Support Foundation of the State of Rio Grande do Sul (FAPERGS), under grant 17/2551-0001206-2.

RESUMO. As empresas buscam alternativas tecnológicas que proporcionem competitividade para seus processos de negócios. Uma delas é a integração de plataformas, ferramentas de software que constroem soluções de integração, que permitem que os diferentes aplicativos que compõem o ecossistema de software trabalhem de forma síncrona e que novas aplicações ou funcionalidades sejam incorporadas com o menor impacto nas existentes. O motor de execução é o componente da plataforma de integração responsável pelo gerenciamento dos recursos computacionais que executam a solução de integração. Entre esses recursos estão as unidades de processamento, chamadas de *threads*, ou de conjuntos de *threads*, chamados *pool de threads* O desempenho do motor de execução está diretamente relacionado ao número de *hreads* disponíveis para executar a solução de integração, mas escalar o número de ≪threads≫ que fornecem um tempo de resposta mais curto é um desafio para os engenheiros de software. Se esta quantidade for subdimensionada, poderá causar um atraso na execução; se for superestimada, poderá causar um desperdício de recursos computacionais. Este artigo apresenta um modelo matemático, definido por equações diferenciais, que estabelece o número ótimo de ≪threads≫, o que maximiza o ganho de desempenho esperado, minimizando o tempo de execução da solução de integração. Além disso, apresenta a aplicação do modelo matemático, que auxilia na análise do ganho esperado em diferentes cenários de arquitetura e quantidade de *thread*.

Palavras-chave: integração de aplicações empresariais, programação *multithread*, motor de execução, modelagem matemática, plataformas de integração.

REFERENCES

[1] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y.S. Ramakrishna & D. White. An efficient meta-lock for implementing ubiquitous synchronization. *Sigplan Notices*, **34**(10) (1999), 207–222.

[2] K. Agrawal, Y. He, W.J. Hsu & C.E. Leiserson. Adaptive Scheduling with Parallelism Feedback. In "Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming" (2006), pp. 100–109.

[3] M. Bhadauria & S.A. McKee. Optimizing Thread Throughput for Multithreaded Workloads on Memory Constrained CMPs. In "Proceedings of the 5th Conference on Computing Frontiers" (2008), pp. 119–128.

[4] E.F. Coutinho, F.R. de Carvalho Sousa, P.A.L. Rego, D.G. Gomes & de José Neuman de Souza. Elasticity in cloud computing: a survey. *Annals of Tecommunications - annales des télécommunications*, **70**(7) (2015), 289–309.

[5] M. Curtis-Maury, J. Dzierwa, C.D. Antonopoulos & D.S. Nikolopoulos. Online Power-performance Adaptation of Multithreaded Programs Using Hardware Event-based Prediction. In "Proceedings of the 20th Annual International Conference on Supercomputing" (2006), pp. 157–166.

[6] A. da Silva Dias, L.H.V. Nakamura, J.C. Estrella, R.H.C. Santana & M.J. Santana. Providing IaaS resources automatically through prediction and monitoring approaches. *IEEE Symposium on Computers and Communications*, (2014), 1–7.

[7] L. Dagum & R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, **5**(1) (1998), 46–55.

[8] W. Dawoud, I. Takouna & C. Meinel. Elastic VM for rapid and optimum virtualized resources allocation. In "5th International DMTF Academic Alliance Workshop on Systems and Virtualization Management: Standards and the Cloud" (2011), pp. 1–4.

[9] Y. Ding, M. Kandemir, P. Raghavan & M.J. Irwin. Adapting Application Execution in CMPs Using Helper Threads. *Journal of Parallel and Distributed Computing*, **69**(9) (2009), 790–806.

[10] D. Dossot, J. D'Emic & V. Romero. "Mule in action". Manning Publications Co. (2014).

[11] E. Ebrahimi, C.J. Lee, O. Mutlu & Y.N. Patt. Fairness via Source Throttling A Configurable and High-performance Fairness Substrate for Multi-core Memory Systems. *Sigplan Notices*, **45**(3) (2010).

[12] K.B. Ferreira, P. Bridges & R. Brightwell. Characterizing Application Sensitivity to OS Interference Using Kernel-level Noise Injection. In "Proceedings of the ACMIEEE Conference on Supercomputing" (2008), pp. 1–12.

[13] M. Fisher, J. Partner, M. Bogoevice & I. Fuld. "Spring integration in action". Manning Publications Co. (2014).

[14] R.Z. Frantz & R. Corchuelo. A software development Kit to implement integration Solutions. In "Proceedings of the 27th Annual ACM Symposium on Applied Computing" (2012), pp. 1647–1652.

[15] R.Z. Frantz, R. Corchuelo & C. Molina-Jiménez. A proposal to detect errors in Enterprise Application Integration solutions. *Journal of Systems and Software*, **85**(3) (2012), 480–497.

[16] R.Z. Frantz, R. Corchuelo & F. Roos-Frantz. On the design of a maintainable software development kit to implement integration solutions. *Journal of Systems and Software*, **111** (2016), 89–104.

[17] R.Z. Frantz, A.M.R. Quintero & R. Corchuelo. A domain-specific language to design enterprise application integration solutions. *International Journal of Cooperative Information Systems*, **20**(02) (2011), 143–176.

[18] D.L. Freire, R.Z. Frantz & F. Roos-Frantz. Ranking enterprise application integration platforms from a performance perspective: An experience report. *Software: Practice and Experience*, **49**(5) (2019), 921–941.

[19] D.L. Freire, R.Z. Frantz, F. Roos-Frantz & S. Sawicki. Survey on the run-time systems of enterprise application integration platforms focusing on performance. *Software: Practice and Experience*, **49**(3) (2019), 341–360.

[20] C. Ibsen & J. Anstey. "Camel in action". Manning Publications Co. (2010).

[21] K. Indrasiri. "Introduction to WSO2 ESB". Springer (2016).

[22] C. Intel. Threading Methodology: Principles and Practices (2010). URL `https://software.intel.com/en-us/articles/threading-methodology-principles-and-practice/`. Last accessed on 01/10/2018.

[23] C. Intel. Get Faster Performance For Many Demanding Business Applications (2018). URL `https://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html/`. Last accessed on 01/12/2018.

[24] C. Jung, D. Lim, J. Lee & S. Han. Adaptive Execution Techniques for SMT Multiprocessor Architectures. In "Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming" (2005), pp. 236–246.

[25] J. Korinth, D. de la Chevallerie & A. Koch. An Open-Source Tool Flow for the Composition of Reconfigurable Hardware Thread Pool Architectures. In "Proceedings of the IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines" (2015), pp. 195–198.

[26] R. Kumar, D.M. Tullsen, P. Ranganathan, N.P. Jouppi & K.I. Farkas. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In "Proceedings of the 31st Annual International Symposium on Computer Architecture" (2004), pp. 64–76.

[27] J. Lee, H. Wu, M. Ravichandran & N. Clark. Thread Tailor: Dynamically Weaving Threads Together for Efficient, Adaptive Parallel Applications. *ACM SIGARCH Computer Architecture News*, **38**(3) (2010), 270–279.

[28] J. Li & J. Martinez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In "Proceedings of the 12th International Symposium on High-Performance Computer Architecture" (2006), pp. 77–87.

[29] H. Linfeng, G. Yuhai & W. Juyuan. Design and implementation of high-speed server based on dynamic thread pool. In "Proceedings of the IEEE 13th International Conference on Electronic Measurement and Instruments" (2017), pp. 442–445.

[30] Y. Ling, T. Mullen & X. Lin. Analysis of optimal thread pool size. *ACM SIGOPS Operating Systems Review*, **34**(2) (2000), 42–55.

[31] A. Lorenzon, M. Cera & A. Beck. Investigating Different General-purpose and Embedded Multicores to Achieve Optimal Trade-offs Between Performance and Energy. *Journal of Parallel and Distributed Computing*, **95**(C) (2016), 107–123.

[32] J. Nieplocha, A. Márquez, J. Feo, D. Chavarría-Miranda, G. Chin, C. Scherrer & N. Beagley. Evaluating the Potential of Multithreaded Platforms for Irregular Scientific Computations. In "Proceedings of the 4th International Conference on Computing Frontiers" (2007), pp. 47–58.

[33] K.K. Pusukuri, R. Gupta & L.N. Bhuyan. Thread Reinforcer Dynamically Determining Number of Threads via OS Level Monitoring. In "Proceedings of the IEEE International Symposium on Workload Characterization" (2011), pp. 116–125.

[34] I. Pyarali, M. Spivak, R. Cytron & D.C. Schmidt. Evaluating and Optimizing Thread Pool Strategies for RealTime CORBA. In "Proc. of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Embedded Systems" (2000), pp. 214–222.

[35] S. Saini, J. Chang, R. Hood & HaoqiangJin. A scalability Study of Columbia using the NAS Parallel Benchmarks. *Computational Methods in Science and Technology*, **SI(1)** (2006), 33–45.

[36] H. Schildt & D. Coward. "The Complete Reference, Tenth Edition". McGraw-Hill Education (2017).

[37] D.C. Schmidt. Evaluating architectures for multithreaded object request brokers. *Communications of the ACM*, **41**(10) (1998), 54–60.

[38] J. Schwarzrock, A. Lorenzon, P. Navaux, A. Beck & E.P. de Freitas. Potential Gains in EDP by Dynamically Adapting the Number of Threads for OpenMP Applications in Embedded Systems. In "VII Brazilian Symposium on Computing Systems Engineering" (2018), pp. 79–85.

[39] K. Singh, M. CurtisMaury, S.A. McKee, F. Blagojević, D.S. Nikolopoulos, B.R. de Supinski & M. Schulz. Comparing Scalability Prediction Strategies on an SMP of CMPs. In "Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part I" (2010), pp. 143–155.

[40] N.D. Singpurwalla & S.P. Wilson. "Statistical methods in software engineering: reliability and risk". Springer Science & Business Media (2012).

[41] M.A. Suleman, M.K. Qureshi & Y.N. Patt. Feedback-driven Threading: Power-efficient and High-performance Execution of Multi-threaded Workloads on CMPs. *ACM SIGARCH Computer Architecture News*, **36**(1) (2008), 277–286.

[42] L.M. Surhone, M.T. Timpledon & S.F. Marseken. "Petals EBS". Betascript Publishing (2010).

[43] M.D. Syer, B. Adams & A.E. Hassan. Identifying Performance Deviations in Thread Pools. In "Proceedings of the 27th IEEE International Conference on Software Maintenance" (2011), pp. 83–92.

[44] A. Tanenbaum & H. Bos. "Modern operating systems". Pearson Inc. (2015).

[45] R. Thekkath & S.J. Eggers. Impact of Sharing-based Thread Placement on Multithreaded Architectures. *SIGARCH Computer Architecture News*, **22**(2) (1994), 176–186.

[46] D. Tsafrir, Y. Etsion, D.G. Feitelson & S. Kirkpatrick. System Noise, OS Clock Ticks, and Fine-grained Parallel Applications. In "Proceedings of the 19th Annual International Conference on Supercomputing" (2005), pp. 303–312.

[47] R. van der Pas. The OMPlab on Sun Systems (2007). URL `http://www.compunity.org/events/upcomingevents/iwomp2007/sw/iwomp2007omplabsunv3.pdf`. Last accessed on 01/14/2018.

[48] Q. Wu, M. Martonosi, D.W. Clark, V.J. Reddi, D. Connors, Y. Wu, J. Lee & D. Brooks. A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance. In "Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture" (2005), pp. 271–282.

[49] D. Xu & B. M.Bode. Performance Study and Dynamic Optimization Design for Thread Pool Systems. In "Proceedings of the International Conference on Computing, Communications, and Control Technologies" (2004).

[50] H. Zhou, L.S. Powers & J. Roveda. Increase the concurrency for multicore systems through collision array based workload assignment. In "Proceedings International Conference on Information Science, Electronics and Electrical Engineering", volume 2 (2014), pp. 1209–1215.

[51] S. Zhuravlev, S. Blagodurov & A. Fedorova. Addressing Shared Resource Contention in Multicore Processors via Scheduling. *Sigplan Notices*, **45**(3) (2010), 129–142.