Online ISSN 1848-3380, Print ISSN 0005-1144
ATKAFF 54(4), 495–506(2013)

Mirko Randić, Bruno Blašković, Šandor Dembitz

Analysis of Sojourn Times in QBD Model of a Thread Pool

DOI 10.7305/automatika.54-4.465
UDK 004.258
IFAC 2.6; 3.2.2

Original scientific paper

Modern Web or database servers are usually designed with a thread pool as a major component for servicing. Controlling of such servers, as well as defining adequate resource management policies, with the aim of minimizing requests' sojourn times presuppose the existence of performance models of thread-pooled systems. In this paper a queuing model of a thread pool is formulated along with a set of underlying assumptions and definitions used. Requests are abstracted in such a way that they are characterized by service time distribution and CPU consumption parameter. The model is defined as a Quasi-Birth-and-Death (QBD) process. Stability conditions for the model are derived and an analytic method based on generating functions for calculation of expected sojourn times is presented. The analytical results thus obtained are evaluated in a developed experimental environment. The environment contains a synthetic workload generator and an instrumented server application based on a standard Java 7 ThreadPoolExecutor thread pool. Sojourn time measurements confirm the theoretical results and also give additional insight into sojourn times related to more realistic workload cases that otherwise would be difficult to analyze formally.

Key words: Thread pool, Sojourn time, CPU-bound tasks, Quasi-Birth-and-Death process, Server application

Analiza vremena boravka u QBD modelu thread pool-a. Kod suvremenih web poslužitelja ili poslužitelja baza podataka thread pool najčešće predstavlja glavnu komponentu za posluživanje. Operativno upravljanje kao i definiranje odgovarajućih politika upravljanja resursima s ciljem minimiziranja vremena boravka zahtjeva u sistemu, pretpostavlja postojanje modela performansi sistema koji sadrže thread pool. U ovom radu je formuliran model thread poola zajedno s definicijama i pretpostavkama na kojima se taj model temelji. Zahtjevi su karakterizirani apstraktno preko raspodjele vremena posluživanja i parametra upotrebe CPU. Model je definiran kao Quasi-Birth-and-Death (QBD) proces. Izvedeni su uvjeti stabilnosti modela i razrađena je metoda proračuna očekivanog vremena boravka zahtjeva. Dobiveni analitički rezultati su provjereni u eksperimentalnoj okolini. Ta okolina se sastoji od generatora opterećenja i instrumentalizirane poslužiteljske aplikacije koja sadrži standardni Java 7 ThreadPoolExecutor. Mjerenja vremena boravka potvrđuju teoretske rezultate i daju nam dodatan uvid u trajanje boravaka koje se odnosi na slučejeve opterećenja koja više odgovaraju praksi, a koje je inače teško ili nemoguće analitički odrediti.

Ključne riječi: thread pool, vrijeme boravka, CPU zahtjevni poslovi, Quasi-Birth-and-Death proces, poslužiteljska aplikacija

1 INTRODUCTION

An increasing number of services are available over the Internet. Servers are expected to process huge numbers of requests concurrently without noticeable degradation of performance (response times and throughput). Processing a request submitted by a client typically involves several steps: performing net (socket) I/O to read the request, analyzing the request, performing a local file I/O or looking up one or several external databases to find relevant information, doing some computations on the results of the queries, dynamically generating a response to the request, and sending the response back to the client via socket I/O. Therefore, requests represent processing tasks that can be further broken down into subtasks characterized by the resources they consume. By applying a level of abstraction to described steps we can say that processing a request involves a mix of computation and waiting, i.e. *CPU-bound* and *non-CPU-bound* subtasks. In modern server systems much more control over resource management is being delegated to the application level, and adequate performance at the application level represents an important factor in overall quality of service. The rationale is that modern server applications themselves (like those for supporting multimedia services, interactive Web

services for end users, database services, real time computing, etc.) are in a better position to know their precise requirements and may dynamically and autonomously adapt their demands to available resources. The research described in this paper has been directed towards analyzing the efficiency that task-handling threads organized in a pool manifest in processing requests characterized by different values of the CPU consumption factor. We define the *CPU consumption factor* r as a parameter ($0 \leq r \leq 1$) that expresses which proportion of an overall request service time belongs to pure CPU processing. The remainder of the time is related to various kinds of waiting. A special situation occurs when processing steps have to be synchronized in a mutually exclusive way. Generally, synchronization introduces additional waiting that can be taken as an opportunity to increase utilization of threads and decrease sojourn times. Understanding of the impact of request characteristics on server performance, e.g. sojourn times, is an important basis for thread pool dimensioning and designing efficient control mechanisms.

The goal of the performance analysis presented in this paper is to find out and understand the impact of the number of threads c in a pool, the CPU consumption factor r, and the synchronization of non-CPU-bound subtasks on sojourn time T_{soj} of requests in a thread pool. We analyze and evaluate the extent to which these application-level multithreading mechanisms can influence server performance in the context of different types of workloads. The performance analysis has been done in two ways: by deriving an analytic performance model for the system and by load testing and sojourn time measurement on a real, instrumented, thread-pooled server application. Such an approach enables us to compare results obtained by the model with those obtained by measurement on running software. We believe that our effort at formal performance modeling represents a contribution toward better understanding thread pool behavior.

The structure of the paper is as follows. First we give an introduction to the problem. Next we discuss related work and some known analytical models. In Section 3 a queuing model of a thread pool is formulated along with a set of assumptions and definitions used. The model is recognized as a quasi-birth-death (QBD) process. Stability conditions of the system are derived, and a formal approach is presented to calculate expected sojourn time that requests spend in the pool, based on generating functions. Results obtained by analysis of the model are evaluated through measurements on an instrumented thread-pooled server. The experimental environment is described in Section 4. It consists of a synthetic workload generator and a server application implemented with a Java 7 ThreadPoolExecutor thread pool from a standard java.lang.concurrency package. Finally, we draw a conclusion.

2 RELATED WORK

Behavior inherent to multithreaded applications includes contention for software resources - threads. On the other hand, threads further contend for and use resources at the lower level: CPUs, memory, I/O communication channels, etc. Therefore, multithreaded applications can be modeled as a two-layered queuing network, representing the software and the hardware layer. The method of layers for analyzing software systems was introduced by Rolia and Sevcik [1]. A multithreaded processing system can also be modeled as a standard queue with FCFS or PS discipline if the number of threads in the pool was $c = 1$ or when $c \to \infty$ respectively. For these queues the sojourn time distribution is known precisely; see for example [2]. Although the number of requests in the system may be unbounded (open systems), the number of requests being served simultaneously may not. Therefore, the egalitarian Processor Sharing discipline (PS) is not feasible in practice and is not even desirable because of significant overhead due to the context switching effect when the number of threads is large. When the number of threads is limited (but larger than 1), this situation is modeled as a queue with the so-called limited processor sharing discipline with c service positions (LPS-c). In practice this means that, when there are c threads active at some moment in time, then each of these c threads receives a fair share $1/c$ of the total CPU capacity. Therefore, the more threads that are active, the smaller is the processor capacity that can be assigned to each thread. In this way, the thread is no longer an autonomous entity operating at a fixed rate; instead, the processing rate of each thread continuously changes over time. Despite its wide range of applications, there are not too many studies on the LPS queue. Avi-Itzhak and Halfin provide some insights into the LPS system, and give an approximation for the expected sojourn time assuming Poisson arrivals. In [3] a closed form equation is given for end-to-end response time of a system with a limited number of service positions c that equally share processor capacity. Expected total sojourn time is approximated by a convex combination of the M/G/1 FCFS and the M/G/1 PS queue with adequate weights. The equation is exact if the number of service positions equals 1 or tends to infinity. A computational analysis of limited processor sharing based on matrix geometric methods is performed in Zhang and Lipsky [4,5]. In [6] van der Weij considered sojourn times in a two-layered tandem LPS queue for small values of c. She extended Avi-Itzhak and Halfin's approximation to a tandem of two multi-server queues, in which the active servers share a common resource in a PS fashion. In [7] a processor sharing queue with a limited number of service positions and an infinite buffer is studied. The authors indicate that the expected sojourn time is monotone in c. In PhD thesis [8], a study of fundamental performance questions

about queuing models with shared resources such as stability issues of these models, product-form solutions and scheduling is presented. Approximation formulas for various performance quantities for the LPS queue are derived in [9] and based on diffusion limits.

A number of research efforts have focused on performance analysis, modeling and control of Web servers. Van der Mei *et al.* [10] present an end-to-end queuing model (a tandem queuing network) for the performance analysis of a Web server, encompassing the impact of client workload characteristics, server hardware/software configurations, communication protocols and interconnect topologies. An interesting paper that points in the direction of a self-adaptable multithreaded server is [11]. Gupta and Harchol-Balter consider the problem of dynamic admission control in resource-sharing systems when the job size distribution has high variability with the aim of minimizing the mean response time. In [12] Randić *et al.* have considered the possibility of applying a swarm recruitment mechanism as the control mechanism for dynamic assignment of threads to two thread pools connected in tandem.

The modeling and analysis approach described in the paper by Nawijn [13] had a strong impact on our work. He analyzed a tandem queue with delayed server release. In fact, he analyzed a queuing system that represents a queuing phenomenon occurring at a gasoline station where a customer serves himself when filling the tank, and subsequently, pays at the counter in the shop. But during the sojourn time in the shop the customer's car blocks one of the servers (pumps), which becomes free again when the customer leaves the station. A similar effect, making a thread unavailable for servicing in the first station because of its migration to the second station, occurs in our model of a thread pool (Section 3).

3 THE MODEL

We define the following model to calculate the mean sojourn time of requests with a specified CPU consumption factor in a thread-pooled server under a stationary regime. A *request* is a typed message that contains a description of some task, i.e. job, to be done, along with the data required to complete that task. A *task* is the fundamental unit of work with well-defined boundaries. Most server applications offer a natural choice of a task boundary - individual client requests. Therefore, any request that enters the server represents a task that should be finished. In our model we adopt a simple formulation of a service offered by the server. Each task related to a request is processed through all of its steps by a single *service position*. In the real server, a service position is implemented as a software resource named a task-handling thread, and these threads are organized in a thread pool. Therefore, in this model the terms service position and thread are used as synonyms.

An overall task is divided into two sequenced subtasks characterized as CPU-bound or non-CPU-bound. Processing the non-CPU-bound subtask results in a thread being blocked or waiting for some event that can be, for example, waiting for data requested from an external database or some other kind of request via network, waiting due to synchronization, etc. Non-CPU-bound subtasks can be independent or dependent in the way that they are synchronized with a mutual exclusion type of synchronization.

Requests arrive at the thread pool according to a Poisson process at the rate $\lambda > 0$. Tasks are characterized by two parameters: the overall mean service time T_s and a parameter r (the CPU consumption part of tasks) specifying the proportion of the overall service time that represents the exclusive CPU processing subtask. Therefore, CPU-bound subtasks have exponentially distributed service times with mean $T_{CPU} = rT_s$ while non-CPU-bound subtasks have exponentially distributed service times with mean $T_{-CPU} = (1 - r)T_s$. The $\mu_1 = 1/T_{CPU}$ and $\mu_2 = 1/T_{-CPU}$ represents corresponding service rates.

The thread-pool model has the form of a tandem queue with two stations (Fig. 1) numbered 1 and 2. Each request demands service at both stations before leaving the system. Stations have buffers with unlimited numbers of waiting positions, i.e. no rejection is possible. The number of service positions occupied or available for processing subtasks in the system is c. The first station is a queue with a limited and variable number of service positions n_1 ($0 \leq n_1 \leq c$) and LPS discipline. If not all threads are occupied, then the buffer is empty and newly arriving requests immediately occupy a thread. Once a request moves into service it is never preempted. Whenever all threads in the first station are occupied, a new arrival joins the end of the thread pool buffer. Once a thread becomes available the first request waiting in the buffer will be served in FCFS order. In the first station, threads are processing CPU-bound subtasks with mean service time T_{CPU}. The processor capacity is always shared equally among all requests occupying service positions in the first station (LPS). After finishing service in the first station, a thread moves to the second station and becomes unavailable for service in the first station. At that moment, the number of service positions in the first station n_1 decrements by one. In the second station, threads are processing non-CPU-bound subtasks with mean service time T_{-CPU}. If the processing of these subtasks is synchronized with mutual exclusion, an arriving thread can either join the synchronization buffer and wait its turn, i.e. wait for other threads to finish their subtasks, or immediately start processing if the second station is empty. The buffer in the second station is filled with threads waiting for the synchronization mechanism, and this is why the maximal number of entities that can be placed in the synchronization buffer is limited to $c - 1$,

i.e. $n_2 \leq c$. If non-CPU-bound subtasks are not synchronized, and if underlying hardware resources responsible for I/O allow true parallelism, then the second station represents an $M/M/n_2$ queue with an empty buffer. After a thread has finished its processing at the second station, it immediately joins threads available for processing in the first station. This is the moment when the number of service positions in the first station n_1 increments and a request's sojourn in the server is completed.

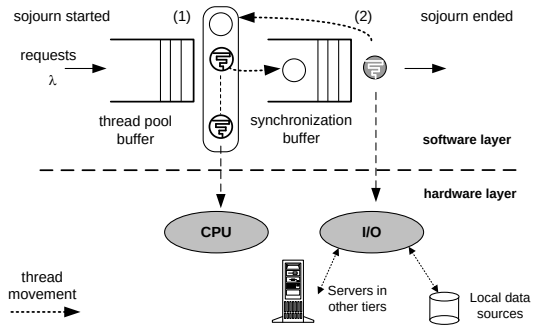

Fig. 1. Thread pool model

The model is completely described by two random variables X_t and Y_t. X_t represents the number of subtasks that are waiting for processing or are being processed in the first station, while Y_t represents the number of subtasks in the second station. By the assumptions mentioned before, the process $\{X_t, Y_t, t \geq 0\}$ is a continuous-time Markov chain with two-dimensional state space $\{0, 1, \cdots\} \times \{0, 1, \cdots, c\}$. A state is represented by a pair (i, j) where the first entry i represents the number of subtasks in the first station at time t. There $i_p = \min\{i, c - j\}$ threads are running CPU-bound subtasks, and the other $i - i_p$ requests for subtasks are waiting in the buffer. The second entry j ($j \leq c$) represents the number of threads in the second station. One thread is running while the other $j-1$ are waiting in the case of synchronized non-CPU bound subtasks or all j threads are running in parallel in the case of unsynchronized subtasks.

Let us assume that the system is in state (i, j). The process evolution is driven by the following transitions (Fig. 2):

a) Arrival of a new request at the first station triggers a transition to state $(i+1, j)$. Arrivals move the process from state (i, j) to state $(i + 1, j)$ at rate λ.

b) A thread that finishes processing at the first station and starts processing at the second station or joins the synchronized queue at the second station, triggers a transition to the state $(i-1, j+1)$. The process jumps from state (i, j) to state $(i - 1, j + 1)$ at service rate μ_1.

c) A thread that finishes processing at the second station joins the first station and triggers a transition to $(i, j - 1)$. The process jumps from state (i, j) to state $(i, j - 1)$ at service rate $m_j\mu_2$, for $j = \{1, \cdots, c\}$. In the synchronized case $m_j = 1\ \forall j$, while in the unsynchronized case $m_j = j$.

For thread pools with $c = 1$ there is no difference between the synchronized and the unsynchronized case because the state transition diagrams are identical and $m_j = 1$ for both cases. In fact this is the case when the synchronization queue has no effect, and the system behaves as an $M/\mathrm{Hypo}(2)/1$ queue with hypo-exponential service time distribution. Only a single request is processed at any given moment.

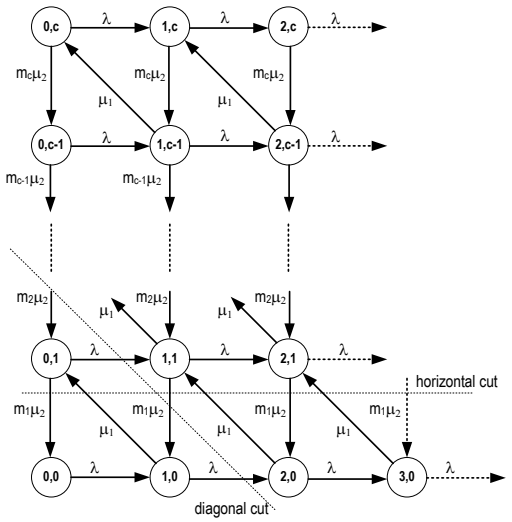

Fig. 2. State transition diagram

Let $P_{i,j} = Prob\{X_t = i, Y_t = j\}$ when $t \to \infty$, then the stationary distribution of the process is denoted by $\{P_{i,j}, i \geq 0, 0 \leq j \leq c\}$. Necessary and sufficient conditions for the existence of the distribution will be stated later. Assuming that a steady state exists, the global balance equations can be expressed as follows:

$$
\begin{aligned}
P_{0,0}\lambda &= P_{0,1}m_1\mu_2 \qquad (1)\\
P_{0,j}(\lambda + m_j\mu_2) &= P_{0,j+1}m_{j+1}\mu_2 + P_{1,j-1}\mu_1\\
&\qquad (1 \leq j \leq c-1)\\
P_{i,0}(\lambda + \mu_1) &= P_{i-1,0}\lambda + P_{i,1}m_1\mu_2\ (i \geq 1)\\
P_{i,j}(\lambda + \mu_1 + m_j\mu_2) &= P_{i-1,j}\lambda + P_{i+1,j-1}\mu_1\\
&\quad + P_{i,j+1}m_{j+1}\mu_2\\
&\qquad (i \geq 1, 1 \leq j \leq c-1)\\
P_{i,c}(\lambda + m_c\mu_2) &= P_{i-1,c}\lambda + P_{i+1,j-1}\mu_1\ (i \geq 1)\\
P_{0,c}(\lambda + m_c\mu_2) &= P_{i+1,c-1}\mu_1
\end{aligned}
$$

3.1 Stability condition for the process

Informally speaking, in an unstable queuing system the total number of requests, under proper scaling in time, will become infinite, whereas a system is stable if the number of requests remains finite. In this subsection we derive stability conditions for the previously formulated continuous time Markov chain $\{X_t, Y_t\}$. It is evident that the chain is an instance of a special class of processes that are called Birth and Death (QBD) processes. The concept of QBD process was introduced by Evans [14] and Wallace [15]. As we emphasized before, the state space S is given as a set of pairs $S = \{(i,j)|0 \leq i, 0 \leq j \leq c\}$. In QBD process theory, the parameter i of the first dimension is called the *level* of the state. The parameter j of the second dimension refers to one of $c+1$ states within a level, which are also called inter-level states or *phases*. By definition, in a QBD process there may be an infinite number of levels, and transitions are only possible between neighboring levels. Levels are divided into two parts: (1) the boundary (or initial) level(s) ($0 \leq i \leq k$); this part must be finite; in our model only level $i = 0$ is a boundary level and (2) the repeating (or repetitive) levels $i \geq k$; this part has to have a regular structure but may be infinite as in our model. The generator matrix of the QBD process is structured as a block-tridiagonal form with repetitive elements:

$$\mathbf{Q} = \begin{bmatrix} B & A_0 & & & & \cdots \\ A_2 & A_1 & A_0 & & & \\ & A_2 & A_1 & A_0 & & \\ & & A_2 & A_1 & A_0 & \\ \vdots & & & & & \ddots \end{bmatrix}$$

where submatrix $\mathbf{A_0}$ encodes forward transitions from level i to level $i+1$, for $i \geq 0$. Submatrix $\mathbf{A_2}$ encodes backward transitions from level i to level $i-1$, for $i > 0$. Finally, submatrices $\mathbf{B}$ or $\mathbf{A_1}$ encode local transitions within level i. The off-diagonal elements of $\mathbf{Q}$ are given by the steady state transition rates as follows: λ – transition rate from level i to level $i+1$ for all phases, μ_1 – transition rate from phase j of an arbitrary level $i > 0$ to phase $j+1$ of level $i-1$, and $m_j\mu_2$ – transition rate from phase $j > 0$ of an arbitrary level i to phase $j-1$ of the same level. The diagonal elements of $\mathbf{Q}$ are given in such a way as to ensure that the elements of each row of $\mathbf{Q}$ sum up to 0. We have defined the following matrices, all of size $[c+1] \times [c+1]$ for boundary $i = 0$ and repeating levels $i \geq 1$:

$$\mathbf{B} = \begin{bmatrix} -\lambda & & & & & \\ m_1\mu_2 & -(\lambda + m_1\mu_2) & & & & \\ & m_2\mu_2 & -(\lambda + m_2\mu_2) & & & \\ & & \ddots & \ddots & & \\ & & & m_c\mu_2 & -(\lambda + m_c\mu_2) \end{bmatrix}$$

$$\mathbf{A_0} = \begin{bmatrix} \lambda & & & \\ & \lambda & & \\ & & \lambda & \\ & & & \ddots \\ & & & & \lambda \end{bmatrix} \quad \mathbf{A_2} = \begin{bmatrix} 0 & \mu_1 & & & \\ & 0 & \mu_1 & & \\ & & 0 & \ddots & \\ & & & \ddots & \mu_1 \\ & & & & 0 \end{bmatrix}$$

$$\mathbf{A_1} = \begin{bmatrix} -(\lambda+\mu_1) & & & & & \\ m_1\mu_2 & -(\lambda+\mu_1+m_1\mu_2) & & & & \\ & m_2\mu_2 & -(\lambda+\mu_1+m_2\mu_2) & & & \\ & & \ddots & \ddots & & \\ & & & m_c\mu_2 & -(\lambda+m_c\mu_2) \end{bmatrix}$$

For reasons of stability and ergodicity the drift of the process to higher levels must be smaller than the drift to the lower levels [16]. Theorem 1.3.2 given in [17] states that any QBD process is positive recurrent if and only if:

$$\pi\mathbf{A_2}\mathbf{e} > \pi\mathbf{A_0}\mathbf{e} \tag{2}$$

where $\mathbf{e}$ is the unit column vector with all entries equal to one and π is the steady state probability vector $\pi = [\pi_0, \pi_1, \cdots, \pi_c]$ of the finite generator matrix $\mathbf{A} = \mathbf{A_2} + \mathbf{A_1} + \mathbf{A_0}$.

Inequality (2) represents a necessary and sufficient stability condition under which the QBD process has a unique stationary distribution $\{P_{i,j}, i \geq 0, 0 \leq j \leq c\}$. The vector π satisfies: $\pi\mathbf{A} = \mathbf{0}$ and $\sum_{j=0}^{c} \pi_j = 1$, making a system of equations that enable us to calculate the probabilities $\pi_0, \pi_1, \cdots, \pi_c$. For different c, stability condition (2) can be expressed as $\lambda < k_c$, where k_c can be computed by the following recursion:

$$\begin{aligned} b_1 &= \mu_1 m_1 \mu_2 \\ n_1 &= \mu_1 + m_1\mu_2 \\ b_{i+1} &= \mu_1 m_{i+1} \mu_2 n_i \\ n_{i+1} &= n_i(\mu_1 + m_{i+1}\mu_2) - b_i \qquad i = 1 \cdots \\ k_i &= \frac{b_i}{n_i} \end{aligned} \tag{3}$$

By replacing $\mu_1 = 1/rT_s$ and $\mu_2 = 1/(1-r)T_s$ in (3) the stability condition becomes $\lambda T_s < k_c(r)$ i.e. $\rho < k_c(r)$. The function $k_c(r)$ represents the boundary load imposed by requests characterized by the CPU consumption factor r that barely imply that a thread pool with c threads will become unstable. If the load is less than $k_c(r)$, the pool remains stable. Functions $k_c(r)$ for synchronized non-CPU-bound subtasks ($m_i = 1\ \forall i$) are calculated and presented in Fig. 3. Pools with $c = 1$ can only accept a maximum load of less than 1 or else factor r becomes relevant. For $c > 1$ maximum load can be applied for tasks with $r = 0.5$. Moreover, $\lim_{c\to\infty} k_c(0.5) = 2$ expresses

the maximum load that can be applied to the system with balanced subtasks ($r = 0.5$) to preserve the stability condition. Functions $k_c(r)$ for unsynchronized non-CPU-bound subtasks ($m_i = i$) are presented in Fig. 4. Maximum load can be applied to the pool when $r = 0$ (pure non-CPU-bound tasks). In this case the pool behaves as an $M/M/c$ system. A more realistic situation is when $r > 0.75$. In that situation, the pool has a similar ability to cope with load, and it is of no consequence whether non-CPU-bound subtasks are synchronized or not.

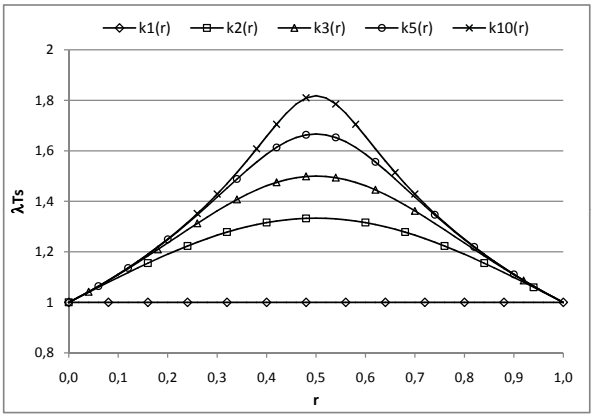

Fig. 3. Allowed maximum load: synchronized non-CPU-bound subtasks

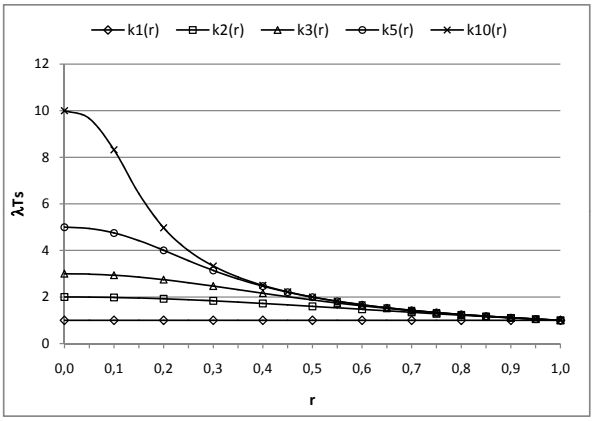

Fig. 4. Allowed maximum load: unsynchronized non-CPU-bound subtasks

3.2 Model analysis: generating function approach

For calculating the average number of requests in the system and their mean sojourn time we used a generating function approach described in the following subsections. Let us define the generating function for each phase of the process:

$$G_j(z) = \sum_{i=0}^{\infty} z^i P_{i,j}, \qquad |z| \leq 1, \qquad 0 \leq j \leq c \quad (4)$$

We use the following analytics to derive generating functions. All those balance equations from (1) that define probabilities $P_{i,j}$ for states related to phase j and all levels $i = 0, \cdots, \infty$ are divided by μ_2. Then we use substitutions $\rho_2 = \lambda/\mu_2$ and $p = \mu_1/\mu_2$. Finally, the equations are multiplied by z^i where i represents corresponding levels. For example, for phase $j = 0$ we consider the following subset of balance equations:

$$\begin{aligned}
P_{00}\lambda &= P_{01}m_1\mu_2 \\
P_{10}(\lambda + \mu_1) &= P_{00}\lambda + P_{11}m_1\mu_2 \\
P_{20}(\lambda + \mu_1) &= P_{10}\lambda + P_{21}m_1\mu_2 \\
&\vdots \\
P_{00}\rho_2 &= P_{01}m_1 \\
P_{10}(\rho_2 + p)z &= P_{00}\rho_2 z + P_{11}m_1 z \\
P_{20}(\rho_2 + p)z^2 &= P_{10}\rho_2 z^2 + P_{21}m_1 z^2 \\
&\vdots
\end{aligned}$$

Summation over i yields:

$$\rho_2 \sum_{i=0}^{\infty} z^i P_{i,0} + p \sum_{i=1}^{\infty} z^i P_{i,0} = \rho_2 \sum_{i=0}^{\infty} z^{i+1} P_{i,0} + m_1 \sum_{i=0}^{\infty} z^i P_{i,1}$$

Multiplying this equation by z and introducing the following notation for compactness:

$$\begin{aligned}
a_0(z) &= \rho_2(1 - z) + p \\
a_j(z) &= \rho_2(1 - z) + m_j + p, \qquad 1 \leq j < c \\
a_c(z) &= \rho_2(1 - z) + m_c
\end{aligned}$$

we obtain the equation for phase 0:

$$z a_0(z) G_0(z) - m_1 z G_1(z) = p P_{00} z$$

By repeating the procedure over all phases $0 \leq j \leq c$ we obtain a system of $c + 1$ linear equations. This is a common result that is a consequence of the application of generating functions. An infinite set of global balance equations (1) has been transformed into a finite set of linear

equations that can be represented as a matrix equation:

$$\begin{bmatrix} za_0(z) & -m_1 z & 0 & \cdots & & 0 \\ -p & za_1(z) & -m_1 z & 0 & & \vdots \\ 0 & -p & za_1(z) & \ddots & & \\ \vdots & 0 & & \ddots & & 0 \\ & & & \ddots & za_{c-1}(z) & -m_c z \\ 0 & \cdots & & 0 & -p & za_c(z) \end{bmatrix} \cdot \begin{bmatrix} G_0(z) \\ G_1(z) \\ G_2(z) \\ \vdots \\ G_{c-1}(z) \\ G_c(z) \end{bmatrix} = \begin{bmatrix} pP_{00}z \\ pP_{01}z - pP_{00} \\ pP_{02}z - pP_{01} \\ \vdots \\ pP_{0,c-1}z - pP_{0,c-2} \\ -pP_{0,c-1} \end{bmatrix} \tag{5}$$

or in short: $\mathbf{A}(z)\mathbf{G}(z) = \mathbf{P}(z)$. $\mathbf{A}(z)$ is a $[c+1] \times [c+1]$ matrix. Unknown generating functions $G_j(z)$ can be calculated from (5) as: $G_j(z) = D_j(z)/D(z), j = 0, \cdots, c$ where $D_j(z) = det\mathbf{A}_j(z)$ and $D(z) = det\mathbf{A}(z)$ are determinants following from Cramer's rule; i.e. $\mathbf{A}_j(z)$ is obtained from $\mathbf{A}(z)$ by replacing the j-th column by $\mathbf{P}(z)$. However, the c "boundary level" probabilities $P_{0,j}$ for $0 \leq j \leq c-1$ have to be determined first. Knowledge of these probabilities fully determines the generating functions.

For calculating boundary level probabilities $P_{0,j}$ for $0 \leq j \leq c-1$ we need c equations in $P_{0,j}$. It is trivial to determine these probabilities if $c \leq 2$ (a single-threaded or double-threaded pool). It can be done from a system of equations consisting of the normalization condition and a global balance equation. If c is greater, we need additional analytics and a more elaborate approach. The approach is based on determining and resolving a system of linear equations composed of: a) the normalization condition, b) an appropriate subset of balance equations, and c) equations in $P_{0,j}$ obtained from zeros of determinant $det\mathbf{A}(z) = 0$ inside or on the unit circle, i.e. where $|z| \leq 1$.

a) The normalization condition is a global equation containing $P_{0,j}$ derived from $c-1$ equations that follow from horizontal cuts on a state transition diagram (see Fig. 2):

$$P_j m_j \mu_2 = P_{j-1}\mu_1 - P_{0,j-1}\mu_1 \qquad 1 \leq j \leq c$$

and divided by μ_2,

$$P_j m_j = pP_{j-1} - pP_{0,j-1} \qquad 1 \leq j \leq c \tag{6}$$

and one equation that is obtained by summation over all diagonal cuts on a state transition diagram:

$$\lambda \sum_{j=0}^{c} P_j = \mu_2 \sum_{j=1}^{c} m_j P_j,$$

divided by μ_2 it yields:

$$\rho_2 = \sum_{j=1}^{c} m_j P_j \tag{7}$$

The set of equations (6) enables us to express all P_j's solely by boundary level probabilities $P_{0,j}$ and P_0 solely. Inserting these P_j's in (7) and respecting that $\sum_{j=0}^{c} P_j = 1$, we obtain the normalization condition:

$$\begin{aligned} & p\left\{\sum_{j=0}^{c-1} P_{0j}\left(\sum_{k=0}^{c-1} p^k \left(\prod_{l=k+1}^{c} m_l\right)\right)\right. \\ & \left. -\sum_{j=1}^{c-1} P_{0j}\left(\sum_{k=c-j}^{c-1} p^k \left(\prod_{l=k-(c-j-1)}^{j} m_l\right)\right)\right\} \\ & = \sum_{k=1}^{c} p^k \left(\prod_{l=k}^{c} m_l\right) - \rho_2 \sum_{k=0}^{c} p^k \left(\prod_{l=k+1}^{c} m_l\right) \end{aligned}$$

For the synchronized case where the $m_l = 1\ \forall l$, the normalization condition becomes:

$$p\sum_{j=0}^{c-1} P_{0,j} \sum_{k=0}^{c-j-1} p^k = \sum_{k=1}^{c} p^k - \rho_2 \sum_{k=0}^{c} p^k$$

b) Additional equations in $P_{0,j}$ can be obtained from zeros of $det\mathbf{A}(z)$. The number of equations depends on the number of roots that the polynomial $det\mathbf{A}(z)$ has in the open interval $(0,1)$. The problem of determining roots is considered in certain papers (see for example Theorem 5.3.1 in [18]). Polynomial $det\mathbf{A}(z)$ (see the form of the matrix $\mathbf{A}(z)$ in equation (5)) has $c/2 - 0.5$ roots for an odd number of threads c, and $c/2 - 1$ roots for an even number of c in the interval $z \in (0,1)$. Also, it has a root of multiplicity $c/2+0.5$ at $z = 0$ for odd c and a root of multiplicity $c/2+1$ at $z = 0$ for even c. Finally, the polynomial has a single root at $z = 1$. If there exists $z_0 \in (0,1)$ such that $det\mathbf{A}(z_0) = 0$, then $det\mathbf{A}_j(z_0)$ for all $j = 0, \cdots, c$ must equal 0 as well because generating functions $G_j(z_0) = det\mathbf{A}_j(z_0)/det\mathbf{A}(z_0)$, by definition, converge in the interval $[0,1]$. Since $G_j(z)$ are analytic in $|z| \leq 1$, $det\mathbf{A}_j(z)$ have to vanish at all those roots where $det\mathbf{A}(z)$ vanishes. Hence, for every $z_0 \in (0,1)$ we obtain an additional equation for the unknown probabilities of the form $det\mathbf{A}_j(z_0) = 0$. Note

that the equations $det\mathbf{A}_j(z_0) = 0$ for $j = 0, \cdots, c$ are linearly dependent [19] and therefore yield only one equation for one root.

c) For $c \geq 2$, to calculate c boundary probabilities $P_{0,j}$ we need to include $c^2/4$ balance equations if c is even, or $(c-1)^2/4$ equations if c is odd. See Table 1 with examples for some c. If $c = 1$, the only boundary probability $P_{0,0}$ can be simply derived from the normalization condition: $m_1 p P_{0,0} = m_1 p - (p + m_1)\rho_2$.

For both the synchronized and unsynchronized case:

$$P_{0,0} = 1 - \rho_1 - \rho_2$$

If $c = 2$, $P_{0,0}$, $P_{0,1}$ can be derived from the normalization condition and one balance equation for $P_{0,0}$:

$$\begin{aligned}(m_1 m_2 p^2 + m_1^2 m_2 p)P_{0,0} \\ + ((m_1 m_2 - m_1^2)p^2 + m_1^2 m_2 p)P_{0,1} \\ = m_1^2 m_2 p + m_1 m_2 p^2 \\ - (m_1 p^2 + m_1 m_2 p + m_1^2 m_2)\rho_2 \\ P_{0,0}\lambda = P_{0,1} m_1 \mu_2\end{aligned}$$

Table 1 shows balance equations for P_{ij} that must be included in the system for calculating boundary probabilities for $c = 2, 3, 4, 5, 10, 11$.

Table 1. Necessary balance equations

$c = 10, 11$	$P_{0,8}$				
	$P_{0,7}$				
	$P_{0,6}$	$P_{1,6}$			
	$P_{0,5}$	$P_{1,5}$			
	$P_{0,4}$	$P_{1,4}$	$P_{2,4}$		
	$P_{0,3}$	$P_{1,3}$	$P_{2,3}$		
$c = 4, 5$	$P_{0,2}$	$P_{1,2}$	$P_{2,2}$	$P_{3,2}$	
	$P_{0,1}$	$P_{1,1}$	$P_{2,1}$	$P_{3,1}$	
$c = 2, 3$	$P_{0,0}$	$P_{1,0}$	$P_{2,0}$	$P_{3,0}$	$P_{4,0}$

3.3 Calculating sojourn times

We have already stressed the following characteristic of generating functions: for all $|z| \leq 1$, $|G_j(z)| \leq 1$, i.e. it is convergent. Furthermore, $G_j(z = 1) = P_j$, where P_j are boundary probabilities: $P_j = \sum_{i=0}^{\infty} P_{i,j}$, for $0 \leq j \leq c$. The long run, i.e. the expected number of requests $E[N] = \sum_{i=0}^{\infty}\sum_{j=0}^{c}(i+j)P_{i,j}$ in the system, can be determined from generating functions $G(z)$. First we calculate the average number of requests N_j related to each phase j of the process.

$$\begin{aligned}G_j(z) &= P_{0,j}z^0 + P_{1,j}z^1 + P_{2,j}z^2 + \cdots \\ z^j G_j(z) &= P_{0,j}z^j + P_{1,j}z^{j+1} + P_{2,j}z^{j+2} + \cdots \\ \frac{d}{dz}\left(z^j G_j(z)\right) &= jP_{0,j}z^{j-1} + (1+j)P_{1,j}z^j \\ &\quad + (2+j)P_{2,j}z^{j+1} + \cdots \\ \lim_{z\to 1}\frac{d}{dz}\left(z^j G_j(z)\right) &= jP_{0,j} + (1+j)P_{1,j} + (2+j)P_{2,j} \\ &\quad + \cdots = \sum_{i=0}^{\infty} P_{i,j}(i+j) = N_j\end{aligned}$$

Therefore,

$$N_j = \lim_{z\to 1}\left(\frac{d}{dz}\left(z^j G_j(z)\right)\right) = \lim_{z\to 1}\left(\frac{d}{dz}\left(z^j \frac{D_j(z)}{D(z)}\right)\right) = \lim_{z\to 1}\left(\frac{jz^{j-1}D_j(z)D(z) + z^j D_j'(z)D(z) - z^j D_j(z)D'(z)}{D^2(z)}\right)$$

and applying the L'Hospital rule yields:

$$N_j = \lim_{z\to 1}\frac{2jD_j'D' + D_j''D' - D_j'D''}{2(D')^2}$$

Finally: $E[N] = \sum_{j=0}^{c} N_j$ can be used to calculate mean sojourn time by Little's law as: $E[T_{soj}] = E[N]/\lambda$. The calculated mean sojourn time of requests in a thread pool both for the synchronized and unsynchronized case is depicted in diagrams in Fig. 5 and Fig. 6. It is worth mentioning that in both cases the model gives similar results when $r > 0.75$, and this range of values characterizes requests that arrive in real server applications.

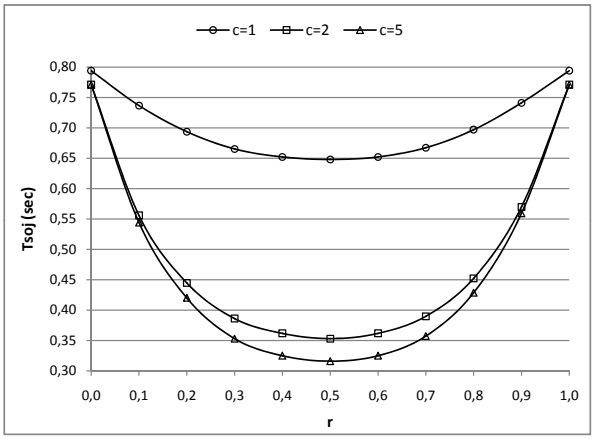

Fig. 5. Calculated sojourn time (synchr. case)

All determinant calculation, analysis of the roots, derivations together with limits calculation necessary for calculation analytics described in this Section, and results depicted in diagrams have been done with the Maxima tool [20].

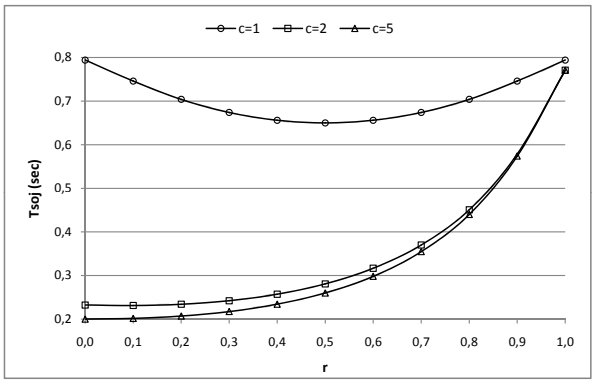

Fig. 6. Calculated sojourn time (unsynchr. case)

4 EXPERIMENTAL SETUP

To evaluate analytical results, we have established an experimental environment based on Java technology. The environment consists of a server and a client program that are started on separated PCs. The server is materialized as an Intel Core i7 2.4GHz CPU with 8 cores. Cores can be disabled as needed. The client program mimics a generator of user requests. The computers were connected with HTTP over 100Mb/s of isolated LAN that was free from additional traffic. Both the thread-pooled server and the client were written in Java (JDK1.7).

Performance analysis consisted of measurements of sojourn times. The application's code on the server was instrumented for measurement times that the threads are spending processing requests. All other activities on the server system, such as background processes and other applications, affect measurements, so experiments should be done carefully on an unloaded system, and results averaged across a number of repeated measurements.

The thread-pooled server application is intentionally designed to be simple, so that no application-related factors other than settings of the thread-pool parameters and request characteristics can significantly influence the performance. To implement the thread pool we have used the ThreadPoolExecutor component with the ArrayBlockingQueue buffer from a standard java.util.concurrent package. Places in the thread-pool buffer were always set to be large to avoid rejections. The design of the server application fits a common multithreading scheme. The daemon (connection) thread waits for requests on a socket and puts them into the thread-pool buffer. Task-handling threads are waiting for a new request to be inserted into the buffer. After finishing execution of a task, a thread returns to the pool and tries to get new work to do. If all threads are busy when a request arrives, the start of the execution is delayed (the request is waiting in the buffer) until a task-handling thread becomes free.

Achieving both precise duration of task processing and measurement precision was not easy. Special care was taken to achieve precision in duration of processing. Processing of subtasks should last as close as possible to the periods specified by each request. To achieve this, we have modified standard java.lang.Thread and used it as a task-handling thread in the pool. We made modifications directly to the Thread class, not via class inheritance to be assured that we did not introduce any additional burden due to the inheritance mechanism, method overriding and interactions between thread-pool object and threads that could affect results of measurement. More precisely, we added to the Thread class a new method cpuBoundTask and a few attributes. We modified the standard sleep method too. Method with prototype:

```
void cpuBoundTask(
    long cpuProcessingTime,
    ThreadMXBean tmxBean)
```

is designed to occupy the CPU for a duration as close as possible to the period specified by parameter cpuProcessingTime in milliseconds. Of course, the thread will be occupied longer, i.e. real (wall) time is greater then cpuProcessingTime, if more than one thread in the pool contend for the CPU. While the cpuBoundTask method is running, elapsed CPU times are measured by the ThreadMXBean. These measurements are not affected by other activities. CPU time is the total time an application or part of it (e.g. a thread) spent using CPU. A call to the modified sleep method emulates waiting for a service on another machine (e.g. a database server) Calls to sleep can be synchronized or unsynchronized.

The real constraint in the server design was the time measurement imprecision implied by the OS. For MS Windows, time intervals less than 17 ms cannot be measured accurately at the application level. Therefore, in both cpuBoundTask and sleep methods a time correction algorithm was included. Any processing that lasts more or less then requested, causes an adequate correction to processing time that is transferred and applied to the next request. With the above-mentioned design improvements to the experimental setup, accurate measurement can be obtained if quantities of time have means greater than 50 (ms).

The experimental setup allows the evaluation of analytical results presented in Section 3. Furthermore, the environment allows extending measurements to requests that had service times distributed with a squared coefficient of variation c_v greater than one. Also, the environment allows analyzing the impact that usage of multiple cores has on sojourn times. The assumption of greater c_v-s better matches the characteristics of real Web requests. Diagrams in this section represent measured so-

journ times in the thread-pooled server with: a fixed number c of threads, Poisson arrivals of requests with intensity $\lambda = 1/0.267$ (req/sec) and exponential ($c_v = 1$) or gamma ($c_v > 1$) service times, and distributions of CPU- and non-CPU-bound subtasks with means: $T_{CPU} = rT_s$ and $T_{-CPU} = (1-r)T_s$. The overall service time was $T_s = 0.2$ (sec) and the load $\rho = \lambda T_s = 0.75$.

Sojourn times measured on a synchronized server with $c = 3$ and different squared coefficients of variations of the task's service times are presented in Fig. 7. Sojourn time grows with c_v especially for tasks with lower r.

Only for $c_v = 1$ is the curve symmetric. But, for $c_v > 1$ the system is more effective if tasks with a greater CPU consumption factor are processed. We have already mentioned that for boundary cases $r = 0$ and $r = 1$ a synchronized thread pool behaves as an ordinary $M/G/1$ FCFS and an $M/G/1$ LPS-c queue respectively. We recall that PS discipline is more effective than FCFS in the cases where service time distributions have greater variations i.e. when $c_v > 1$. The curve for $c = 3$ and $c_v = 1$ fits those obtained analytically (depicted earlier in Fig. 5).

For unsynchronized non-CPU-bound subtasks the system is more effective when $c > 1$; see curves with $c = 3$ in Fig. 8. For the boundary case where $r = 0$, the system behaves as an $M/G/c$ FCFS queue that is much more effective if compared with LPS-c. Curves with $c_v = 1$ match very well to those obtained analytically (depicted earlier in Fig. 6).

Curves in Fig. 9 can be compared with those in Fig. 5 because they represent the same thread-pool configuration but loaded by requests that impose service times with a greater coefficient of variation $c_v = 3$. Curves become asymmetric and the system displays weaker performance in regard to the sojourn times.

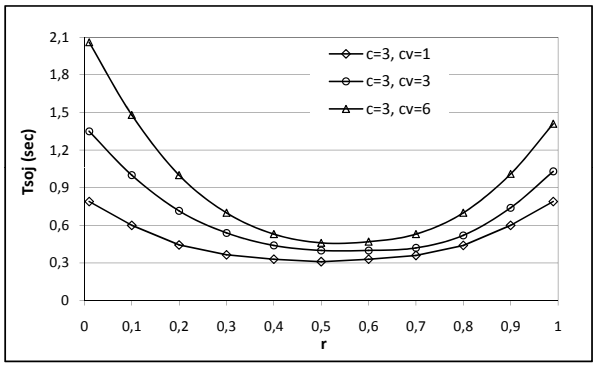

Fig. 7. Measured T_{soj} (synchr. case)

Curves in Fig. 10 represent sojourn times measured on the thread pool while the processing is done by a variable number of cores (1, 2, and 4). The number of threads was

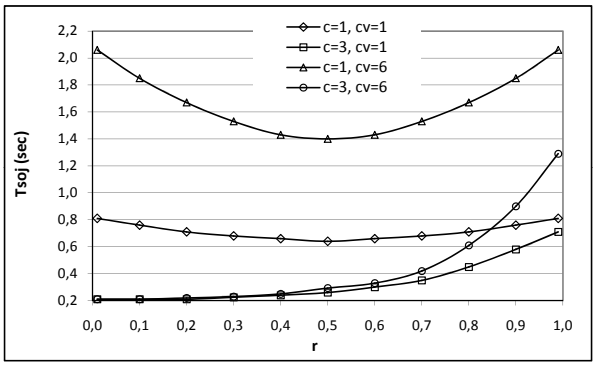

Fig. 8. Measured T_{soj} (unsynchronized case)

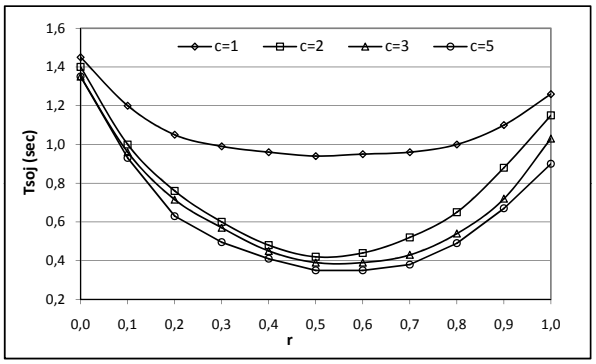

Fig. 9. Effect of c on sojourn times when $c_v = 3$

fixed $c = 2$, and the pool was loaded with requests characterized by $c_v = 3$, variable r and synchronized threads. Loads have remained unchanged $\rho = 0.75$ ($T = 0.20$ sec). The curve with $N_{core} = 1$ is a reference curve, and speedup for $N_{core} > 1$ can be calculated in regard to it. It was expected that, for a load less than one, adding more than two cores will not contribute to the system's performance. Furthermore, speedup is noticeable for requests characterized by $r > 0.5$. It was explained earlier that factor rT represents units of time during which the algorithm for serving requests uses the CPU. It represents the parallelizable part of the algorithm. On the other hand there are requests with $r < 0.5$ representing tasks with only small parallelizable parts. In this range of values for r, speedup is negligible. All discussions related to Fig. 10 are in accordance with Amdahl's law [21].

5 CONCLUSION

Resource sharing systems such as Web or database servers are loaded with requests with different CPU consumption demands and service time distributions. These servers are usually designed and structured with a thread pool as a major component for servicing. Dimensioning and controlling of such servers, as well as defining ade-

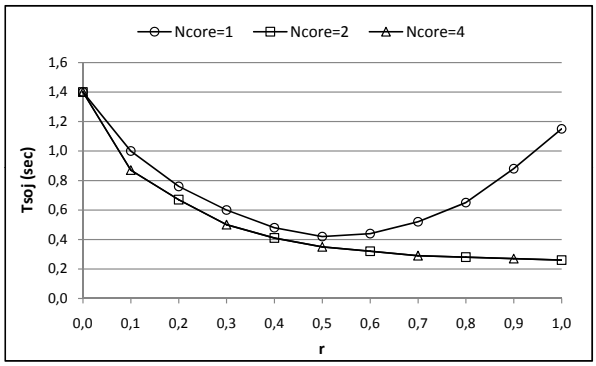

Fig. 10. Speedup due to parallelization

quate resource management policies, with the aim of minimizing requests' sojourn times, all presuppose comprehensive performance analysis of thread-pooled systems. Generally, design of efficient multithreaded programs suited to the next-generation processors will be based on knowledge obtained through the performance analysis of such systems.

We define a model of a thread pool that allows us to calculate the mean sojourn times as a function of system load $(\lambda, \mu_1\mu_2)$, number of threads in the pool c and the CPU consumption factor r of requests. The model assumes that requests' service times are exponentially distributed with a squared coefficient of variation equal to one. The sojourn times related to the requests with exponentially distributed service times predicted by the model fit the experimental outcome very well. Furthermore, the experimental environment should enable us to measure performance metrics on the thread pool with blocking i.e. those pools that have buffers with a limited number of places. These issues are not considered in the paper. Analysis of sojourn times confirms that for both synchronized and unsynchronized cases these times decrease very quickly when the number of threads increases and tasks are balanced. But if tasks are purely CPU- or non-CPU-bound $r \approx 0$ or $r \approx 1$ and service times are exponentially distributed, the number of threads has no effect on sojourn times. Measurement results related to service time distributions with $c_v > 1$ indicate how sojourn time changes with respect to c and c_v.

Most of the results presented in this paper are related to the situation where only one CPU is available for running threads. We believe that the performance model of a thread pool as it is presented and analyzed in this paper represents a valuable basis that can be further developed to comprehend multi-core systems.

REFERENCES

[1] J. A. Rolia and K. C. Sevcik, "The method of layers," *Software Engineering, IEEE Transactions on*, vol. 21, no. 8, pp. 689–700, 1995.

[2] R. W. Wolf, "Stochastic modeling and the theory of queues," *Printice Hall*, 1989.

[3] B. Avi-Itzhak and S. Halfin, "Expected response times in a non-symmetric time sharing queue with a limited number of service positions," in *Proceedings of ITC*, vol. 12, pp. 2–1, 1988.

[4] F. Zhang and L. Lipsky, "Modelling restricted processor sharing," in *Proc. of the 2006 Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA06)*, 2006.

[5] F. Zhang and L. Lipsky, "An analytical model for computer systems with non-exponential service times and memory thrashing overhead," in *Proc. of the 2007 Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA07)*, 2007.

[6] W. van der Weij, "Sojourn times in a two-layered tandem queue with limited service positions and a shared processor," tech. rep., Faculty of Economics and Econometrics, University of Amsterdam, 2004.

[7] M. Nuyens and W. v. d. Weij, "Monotonicity in the limited processor-sharing queue," *Stochastic Models*, vol. 25, no. 3, pp. 408–419, 2009.

[8] W. van der Weij, *Queueing Networks with Shared Resources*. PhD thesis, Vrije Universiteit, Amsterdam, 2009.

[9] J. Zhang and B. Zwart, "Steady state approximations of limited processor sharing queues in heavy traffic," *Queueing Systems*, vol. 60, no. 3-4, pp. 227–246, 2008.

[10] R. D. van der Mei, R. Hariharan, and P. Reeser, "Web server performance modeling," *Telecommunication Systems*, vol. 16, no. 3-4, pp. 361–378, 2001.

[11] V. Gupta and M. Harchol-Balter, "Self-adaptive admission control policies for resource-sharing systems," in *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, pp. 311–322, ACM, 2009.

[12] M. Randić, H. Jednaković, and B. Blašković, "Dynamic thread assignment in a tandem of thread pools inspired by the adaptation mechanism in honeybee foraging," in *Proceedings of the 21st International DAAAM Symposium*, vol. 21, pp. 47–49, 2010.

[13] W. M. Nawijn, "A tandem queue with delayed server release," tech. rep., Universiteit Twente, 1997.

[14] R. Evans, "Geometric distribution in some two-dimensional queueing systemsgeometric distribution in some two-dimensional queueing systems," in *Operation Research*, vol. 15, pp. 830–846, 1967.

[15] V. L. Wallace, *The Solution of Quasi Birth and Death Processes Arising from Multiple Access Computer Systems*. PhD thesis, University of Michigan, 1969.

[16] M. F. Neuts, *Matrix-geometric solutions in stochastic models: an algorithmic approach*. Courier Dover Publications, 1981.

[17] M. F. Neuts, "Structured stochastic matrices of m/g/1 type and their applications," *Marcel Decker Inc., New York*, 1989.

[18] V. Kitsio and U. Yechiali, "Multi-server queues with intermediate buffer and delayed information on service completions," *Stochastic Models*, vol. 24, no. 2, pp. 212–245, 2008.

[19] Y. Levy and U. Yechiali, "An m/m/s queue with servers' vacations," *Information Systems and Operational Research*, vol. 14, no. 2, pp. 153–163, 1976.

[20] R. H. Rand, "Introduction to maxima," *Cornel University*, vol. 25, 2005.

[21] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, pp. 483–485, ACM, 1967.

Mirko Randić (1962) received B.Sc., M.Sc. and Ph.D. degrees in electrical engineering from the University of Zagreb, Faculty of Electrical Engineering and Computing, Zagreb, Croatia, in 1985, 1988 and 1998, respectively. Since 1986 he has been working at the Faculty of Electrical Engineering and Computing, Zagreb. His current position is the assistant professor at Department of Electrical Engineering Fundamentals and Measurements. Currently he is a researcher on the projects: "Knowledge-Based Network and Service Management" and "Networked Economy" financed by the Ministry of science, education and sports of the Republic of Croatia. He published more than 40 papers in journals and conference proceedings in the area of networks and service management, software systems modeling and service performance models. Doc. Randić is a member of IEEE.

Bruno Blašković received B.Sc., M.Sc. and Ph.D. degrees in electrical engineering from the University of Zagreb, Faculty of Electrical Engineering and Computing (FER), Zagreb, Croatia, in 1982, 1985 and 1996, respectively. From December 1986 he is working at the Department of Electrical Engineering Fundamentals and Measurements at FER. In December 2003 he was promoted to professor. He published over 60 papers in journals and conference proceedings. His current research interests are in the field of protocol synthesis, model transformations, business process modeling, formal methods, software testing, model checking and network reliability. Prof. Blašković is also a member of several international professional associations. He participated in conference international programs committees, and he serves as a technical reviewer for international journals.

Šandor Dembitz (1951) received B.Sc., M.Sc. and Ph.D. degrees in electrical engineering from the University of Zagreb, Faculty of Electrical Engineering and Computing (FER), Zagreb, Croatia, in 1973, 1981 and 1993, respectively. From February 1974 he is working at the Department of Fundamentals of Electrical Engineering and Measurement at FER. In October 2004 he was promoted to Associate Professor. He participated in many national scientific projects and two COST actions. Currently he is a researcher on the project "Networked Economy" financed by the Ministry of Science, Education and Sports of the Republic of Croatia, and COST Action IC1002 „Multilingual and Multifaceted Interactive Information Access" (MUMIA). Prof. Dembitz is author and maintainer of Hascheck, 20 years old Croatian online spellchecker, which became a world-wide popular service used in more than 100 countries. He published more than 100 papers in journals and conference proceedings in the area of natural language processing, expert systems, information system modeling and service performance models (Hascheck). Prof. Dembitz is a member of IEEE.

AUTHORS' ADDRESSES

Asst. Prof. Mirko Randić, Ph.D.
Prof. Bruno Blašković, Ph.D.
Prof. Šandor Dembitz, Ph.D.
Faculty of Electrical Engineering and Computing,
University of Zagreb,
Unska 3, HR-10000, Zagreb, Croatia
email: mirko.randic@fer.hr, bruno.blaskovic@fer.hr, sandor.dembitz@fer.hr

Received: 2013-01-22
Accepted: 2013-05-10