# Optimal Task Assignment in Multithreaded Processors: A Statistical Approach

Petar Radojković
Barcelona Supercomputing Center (BSC), Barcelona, Spain
petar.radojkovic@bsc.es

Vladimir Čakarević
Barcelona Supercomputing Center, Barcelona, Spain
vladimir.cakarevic@bsc.es

Miquel Moretó
Universitat Politècnica de Catalunya (UPC) & BSC, Barcelona, Spain
mmoreto@ac.upc.edu

Javier Verdú
Universitat Politècnica de Catalunya, Barcelona, Spain
jverdu@ac.upc.edu

Alex Pajuelo
Universitat Politècnica de Catalunya, Barcelona, Spain
mpajuelo@ac.upc.edu

Francisco J. Cazorla
BSC & Spanish National Research Council (IIIA-CSIC), Barcelona, Spain
francisco.cazorla@bsc.es

Mario Nemirovsky
ICREA Research Professor at BSC, Barcelona, Spain
mario.nemirovsky@bsc.es

Mateo Valero
BSC & UPC, Barcelona, Spain
mateo@ac.upc.edu

## Abstract

The introduction of massively multithreaded (MMT) processors, comprised of a large number of cores with many shared resources, has made task scheduling, in particular task to hardware thread assignment, one of the most promising ways to improve system performance. However, finding an optimal task assignment for a workload running on MMT processors is an NP-complete problem.

Due to the fact that the performance of the best possible task assignment is unknown, the room for improvement of current task-assignment algorithms cannot be determined. This is a major problem for the industry because it could lead to: (1) A waste of resources if excessive effort is devoted to improving a task assignment algorithm that already provides a performance that is close to the optimal one, or (2) significant performance loss if insufficient effort is devoted to improving poorly-performing task assignment algorithms.

In this paper, we present a method based on Extreme Value Theory that allows the prediction of the performance of the optimal task assignment in MMT processors. We further show that executing a sample of several hundred or several thousand *random* task assignments is enough to obtain, with very high confidence, an assignment with a performance that is close to the optimal one. We validate our method with an industrial case study for a set of multithreaded network applications running on an UltraSPARC T2 processor.

***Categories and Subject Descriptors*** D4 [*Operating systems*]: Process management—Scheduling

***General Terms*** Performance, experimentation

***Keywords*** Scheduling, Task assignment, Multithreading, Statistical estimation, Extreme value theory



*ASPLOS'12*, March 3–7, 2012, London, England, UK.



## 1. Introduction

The main purpose of Lightweight Kernel (LWK) implementations is to provide maximum access to the hardware resources for the applications, as well as providing predictable system performance. In order to achieve these goals, system services are restricted only to the ones that are essential. Furthermore, the services are streamlined, thus reducing the overhead of LWKs to the minimum. Also, LWKs usually apply simplified algorithms for task scheduling and memory management that provide a significant and predictable amount of the processor resources to the running applications.

Dynamic scheduling may potentially vary the amount of processing time made available to applications during their execution, which can significantly affect the performance of HPC applications [26, 42] and reduce the performance provided by commercial network processors. Maximizing the amount of computation power delivered to running parallel applications is critical to achieving high performance and scalability. As a result, many commercial systems already use LWKs with static scheduling, such as CNK [48] in BlueGene HPC systems, and Netra DPS [3, 4] which is mainly used in networking.

Multithreaded processors[1] comprised of several cores, where each core supports several simultaneously running threads, have different *levels of resource sharing* [56]. For example, in a CMP processor where each core supports the concurrent execution of several tasks through SMT, all simultaneously running tasks share global resources such as the last level of cache or the I/O. In addition to this, tasks running in the same core share core resources such as the Instruction Fetch Unit, or the L1 instruction and data cache. Therefore, the way that tasks are assigned to cores determines which resources they share, which, in turn, may significantly affect the system performance. In processors with several levels of resource sharing, task scheduling is comprised of two steps. In the first step, usually called *workload selection*, the OS selects the set of tasks (workload) that will be executed in the processor in the next time slice, from a set of ready-to-run tasks. In the second step, called *task assignment*, each task in the workload is assigned to a hardware context (virtual CPU) of the processor.

[1] In this paper, we use the term "multithreaded processor" to refer to any processor that has support for more than one thread running at a time. Chip Multiprocessors (CMPs), Simultaneous Multithreading (SMT), Coarse-grain Multithreading, Fine-Grain Multithreading processors, or any combination of them are multithreaded processors.

In Massively Multithreaded (MMT) processors with large numbers of cores and several levels of resource sharing that increase in each generation [5], finding a good task assignment becomes an intractable problem. As the number of possible task assignments is vast (*e.g.* $10^{50}$) [18, 20, 33, 44], it is unfeasible to do an exhaustive search in order to find the task assignment with the highest performance. Also, the analytical analysis of optimal task assignment is an NP-complete problem [24].

Therefore, current task assignment approaches can not guarantee that the performance of the predicted best assignment is either the optimal one, or close to it. As the performance of the optimal task assignment for a workload is unknown, the room for improvement of current task assignment techniques cannot be determined. Thus, it is difficult to properly determine the effort needed to invest in the task assignment process. This may lead to overspending if a good task assignment algorithm is constantly improved, or sub-optimal performance if a poor-performing algorithm is not enhanced.

In this paper, we present a method based on Extreme Value Theory (EVT) that allows the prediction of the performance of the optimal task assignment. We also show that, in environments in which the workload infrequently changes, the system designer can simply execute a sample of several hundred or several thousand *random* task assignments of a given workload and measure the performance of each assignment. According to this analysis, there is a very high probability that the performance of the best observed assignment will be close to the optimal system performance. This removes the need for any application profiling or an understanding of the increasingly complex MMT architectures. Unlike other task assignment proposals that use performance predictors to find the best task assignment, our method is application and architecture independent. The method can be applied directly and without any change to any architecture running any set of applications. Our study makes the following contributions:

1. We show that running a sample of several hundred or several thousand random task assignments will most probably capture an assignment within 1% or 2% of the best performing ones.
2. We present a statistical method that, based on the measured performance of a sample of random assignments, estimates the performance of the best task assignment *i.e.* the optimal system performance for a given workload.
3. We present an iterative algorithm that, based on the previous analysis, samples random task assignments until it captures an assignment with the acceptable performance difference with respect to the estimated optimal system performance.

We applied the presented analysis and the iterative tasks-assignment algorithm to an industrial case study in which we scheduled multithreaded network applications running on the UltraSPARC T2 processor. For all five applications used in the study, just several thousand random task assignments were enough to capture an assignment with a performance loss below 2.5% with respect to the estimated optimal system performance (the estimated performance of the best task assignment). When the acceptable performance loss increased to 10%, for all the benchmarks in the suite, running less than 1300 random assignments was enough to provide the required performance. In addition to being architecture and application independent, our method required, in the worst case, around two hours of experimentation on the target systems to find the task assignments with performance very close to the optimal one.

The rest of the paper is organized as follows. Section 2 motivates the need for systematic task assignments and quantifies the number of possible different assignments in modern MMT processors. Section 3 presents a statistical analysis that we use to estimate the optimal system performance based on a small sample of random task assignments. Section 4 describes the experimental environment used in the case study. In Section 5, we present the results of experiments in which we applied the presented analysis to the industrial case study for the assignment of multithreaded network applications. Section 6 presents the related work, while Section 7 lists the conclusions of the study.

## 2. Motivation and background

In this paper, we focus on the problem of task assignment for network applications running on massively multithreaded processors. The importance of network applications increases with the number and the complexity of services that these applications provide and with the tremendous growth of Internet traffic. Providing high performance in networking services is critical to sustain these services and avoid dropping packets, since network processors are saturated by a high network bandwidth that is constantly increasing.

In order to provide high-speed processing and high throughput, network applications have specific hardware and operating system requirements. Massively multithreaded processors are the best alternative to exploit the multiple levels of parallelism exposed by network applications [57]. These processors support simultaneous execution of multiple threads that can process numerous packets at the same time. Network-oriented systems require run-time environments that provide high performance, high-speed packet processing, and execution time predictability. To that end, these systems use low-overhead runtime environments that reduce the performance impact of the system overhead because of execution of management tasks [4].

In networking environments, applications continuously process different packets, repeating a similar processing algorithm for each packet. As the applications running on network processors provide a constant set of services and process different packets in a similar manner, dynamic task scheduling is not essential. In addition to this, the workload running on network processors is usually known beforehand and seldom changes at runtime. Hence, the most promising way to increase the performance of network applications running in MMT processor is by finding a task assignment with a performance that is close to the optimal (close-to-the-optimal task assignment), which is the focus of our study.

Doing optimal task assignment requires a deep knowledge of the resource requirements of each running task and an understanding of how the tasks interact in each shared processor resource. Therefore, it is difficult to determine the optimal task assignment without a previous analysis of a large number of experiments that exponentially increases with the number of processor hardware contexts (virtual CPUs), number of different levels of resource sharing, and number of simultaneously running tasks. The problem becomes even more complex for multithreaded applications. When an application comprises several threads, it is not enough to understand how sharing hardware resources affects the execution time of each thread independently. The designer also has to be aware of the way that the threads communicate and to understand which application threads are the bottlenecks that determine application performance. Currently, task assignment is done in one of two ways:

(1) **Manual assignment:** A skilled designer determines the optimal task assignment based on a detailed analysis of the target architecture and an off-line application profiling. This analysis is complex and its complexity increases with the number of processor hardware contexts, number of levels of resource sharing, and number of simultaneously running tasks. In addition to this, any change in the application or in the hardware platform requires the repetition of the whole analysis. Manual task assignment is not a systematic

approach, its performance depends on the designer's skills, and, in general, does not provide the optimal solution.

(2) **Performance predictors:** Several studies [20, 44, 46] propose approaches that predict the performance of different task assignments for a given workload based on some architecture-dependent heuristics. As the number of all possible task assignments is vast (over $10^{50}$ in current architectures), predicting the performance of all assignments is unfeasible. Therefore, the predictors are used to estimate performance for a sample of assignments, and to determine the best task assignment in the given sample. In addition to this, as predictors introduce an error when estimating the performance of task assignments, the best predicted assignment in the sample may not be the actual best one. Another drawback of most of the currently available performance predictors is that they are not designed with multithreaded applications in mind.

In both manual task assignment and performance predictors, any change in the architecture or applications may require significant extra time and effort to find good task assignments under new conditions. If task assignment is done manually, a change in the set of running applications or target architecture requires the designer to repeat the whole analysis. In the case of performance predictors, the whole process of application profiling and performance estimation has to be re-analyzed, and the prediction algorithm may require significant changes.

**Number of possible task assignments in current MMT processors:** In current multithreaded processors comprised of several cores where each core supports several simultaneously-running tasks, the total number of possible task assignments is vast [18, 20, 33, 44]. This number will grow hugely in future massively multithreaded processors in which the number of cores and number of different levels of resource sharing increase [5]. In order to illustrate this, we study the number of possible assignments when several tasks simultaneously execute on the UltraSPARC T2 processor. The UltraSPARC T2 processor comprises eight cores, and each core contains two hardware pipelines. Each (hardware) pipeline can execute up to four tasks at a time, meaning that the maximum number of simultaneously running tasks is 64. The number of possible task assignments for different workload size is shown in Table 1. The first column of the table shows the number of tasks in the workload. We present results for 3, 6, 9, 12, 15, 18, and 60 tasks. The second column presents the number of different task assignments for a given workload size. When the workload is comprised of 3 tasks, *a*, *b* and *c*, the number of possible task assignments is 11. For example, in one assignment, *a* is executed alone on *Core 0*, while *b* and *c* run inside the same hardware pipeline of Core 1: {*[a][]*}{*[bc][]*} [2]. In the second assignment, *a* executes alone on *Core 0*, and *b* and *c* are distributed among two hardware pipelines of Core 1, {*[a][]*}{*[b][c]*}, *etc.* The third column of the table shows the time needed to execute all possible task assignments assuming that each assignment can be executed in only one second. Finally, in the last column, we present the time needed to predict the performance of all task assignments. We make an optimistic assumption that the performance of a single assignment can be predicted in $1\mu s$ that is in the order of 1000 cycles of a processor running at 1GHz frequency.

Overall, we observe that the number of possible tasks assignments as well as the time needed to execute them grows exponentially with the number of tasks in the workload. The point where running all assignments is unfeasible is reached very fast. For 9 tasks in the workload, the time needed to execute all assignments is 7 days, for 12 tasks, the execution time is more than 15 years. For 60-task workloads that use 94% of hardware contexts of the processor, the time needed to execute all possible task assignments is $1.75\times10^{51}$ years. The results presented in Table 1 clearly show that, in general, running all task assignments is unfeasible, and that an exhaustive search cannot be used to find the optimal system performance for a given workload.

[2] The rest of the cores are not shown for the sake of simplicity.

| Number of tasks | Number of possible task assignments | Time needed to execute all task assignments | Time needed to predict all task assignments |
|---|---|---|---|
| 3 | 11 | 11 seconds | 11 $\mu$s |
| 6 | 1,526 | 25 minutes | 1.5 ms |
| 9 | 591 x $10^3$ | 7 days | 0.6 seconds |
| 12 | 458 x $10^6$ | 15 years | 8 minutes |
| 15 | 600 x $10^9$ | 19 thousand years | 7 days |
| 18 | 971 x $10^{12}$ | 30 million years | 31 years |
| 60 | 550 x $10^{56}$ | 1.75 x $10^{51}$ years | 1.75 x $10^{45}$ years |

**Table 1.** Number of different task assignments for applications running on the UltraSPARC T2 processor

From Table 1, we also see that the prediction of all possible tasks assignments is unfeasible. The last column shows that, for example, for the case of 15-task workloads, the time needed to predict all assignments is 7 days. For 18 tasks, the prediction time is measured in years. Therefore, performance predictors can be used only to determine the close-to-the-optimal task assignment in a limited sample. But, as the number of possible task assignments in modern processors is vast, the effectiveness of performance predictors is questionable [18, 20]; even if we assumed that a performance predictor would be able to find the best assignment in the sample, it would not be clear what the performance difference between the best assignment in the sample and the actual global best assignment (the optimal system performance) would be.

**Importance of knowing the optimal system performance:** Numerous studies present the algorithms for task assignments on multithreaded processors (see Section 6). Given that, in general, running all possible task assignments is unfeasible, several authors [6, 20, 44] verify their proposals with respect to a naive task assignment, in which tasks are randomly assigned to the virtual CPUs of the processor, or Linux-like assignments, in which the number of tasks per core or *scheduling domain* is balanced. It is our position that the evaluation of those proposals could significantly improve if they were also compared to the performance of the optimal task assignment.

We use an example to show how the evaluation of task assignments techniques can be misleading if the optimal system performance is unknown. Figure 1 presents the performance of two multithreaded network benchmarks running on the UltraSPARC T2 processor. Both benchmarks used in the experiment, *IPFwd-intadd* and *IPFwd-intmul*, are based on a generic 3-thread pipelined IP Forwarding network application [44]. Figure 1 shows the results for two instances of the 3-thread benchmark (six threads) simultaneously running on the processor. As, in this specific case, the total number of possible task assignments is around 1500, we can obtain the performance for all assignments. The $X$ axis lists the benchmarks that are used in the experiment and the $Y$ axis shows the performance measured in processed Packets Per Second (PPS). The chart is divided into two parts. In the left part, we only plot the results for *Naive* and the *Linux-like* task assignment. In the right part, we also include the bar that corresponds to the optimal system performance.

Based on the results presented in Figure 1, we want to analyze whether the Linux-like scheduler provides a good performance for *IPFwd-intadd* and *IPFwd-intmul* benchmarks running on the UltraSPARC T2 processor. First, we compare only the Linux-like and Naive scheduler (see the left part of the chart). The improve-

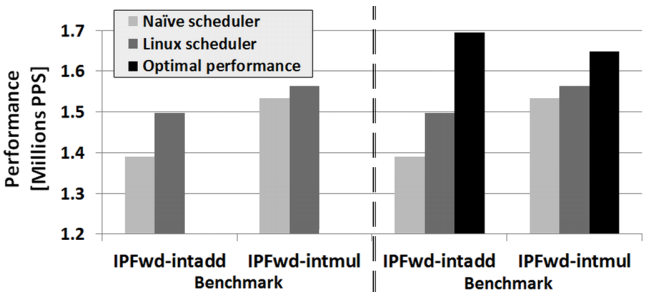



**Figure 1.** Comparison of a naive, Linux-like, and optimal task assignment

ment of the Linux-like scheduler with respect to naive scheduling is around 110,000 PPS (8%) for *IPFwd-intadd* benchmark, and around 30,000 PPS (2%) for *IPFwd-intmul* benchmark. Based on these results, we could conclude that the Linux-like scheduler provides much better performance for *IPFwd-intadd* than for *IPFwd-intmul* benchmark.

However, when we also consider the performance of the optimal task assignment, the conclusions of the analysis change. From the results presented in the right part of the chart, we see that the difference between the optimal performance and performance provided by the naive scheduler is much larger for *IPFwd-intadd* than for *IPFwd-intmul* benchmark, 305,000 PPS (22%) and 115,000 PPS (7%), respectively. Therefore, we conclude that the improvement of the Linux-like scheduler for *IPFwd-intadd* benchmark is higher because the room for the improvement is larger, and not because the Linux-like scheduler fits better with this benchmark. Actually, for the *IPFwd-intmul* benchmark, the Linux-like scheduler provides a performance much closer to the optimal one. For *IPFwd-intmul*, the performance loss of the Linux-like scheduler with respect to optimal performance is only 85,000 PPS (5%). For *IPFwd-intadd*, the performance loss is 200,000 PPS (12%).

Overall, knowing the optimal system performance not only improves the evaluation of different task assignment techniques, but it also shows the performance difference between the proposed task assignments and the optimal one. This performance difference determines the upper limit for improvement of a given task assignment algorithm, which is the most important piece of information for the system designer when deciding whether the algorithm should be enhanced.

## 3. A statistical approach to the task assignment problem

In this paper, we present a statistical method that overcomes the limitations of the approaches currently used for task assignment in multithreaded processors. The analysis is comprised of three parts. First, we execute a sample of random task assignments on the target processor and measure the performance of each one of them. We analyze whether the best performing assignment in the random sample exhibits performance close to the optimal one. In order to do this, we analyze the probability that a sample of randomly selected $n$ task assignments contains at least one from the $P\%$ (*e.g.*, $P = 1$, 2, or 5%) of the best-performing assignments. Second, we show how the cumulative distribution function can be used to identify which portion of all task assignments has good performance. Thirdly, we use Extreme Value Theory to statistically estimate the performance of the best task assignment, *i.e.* the optimal system performance for the given workload. We also use the estimated optimal system performance to determine the possible room for the performance improvement of the proposed task assignment method.

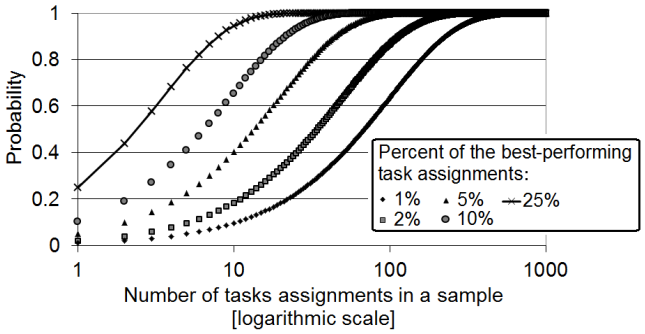



**Figure 2.** Probability that a sample contains a task assignment from P% of the best-performing assignments

### 3.1 Finding task assignments with a good performance

The probability that a sample of random assignments selected from a vast population contains the assignment with *the best* performance is low. However, it is not clear what the probability is that a sample of random assignments contains at least one of the assignments with *a good* performance.

Assume that event $A$ is the probability that a sample contains at least one task assignment from the $P\%$ of best-performing assignment. Event $A'$ is the opposite of event $A$, representing the probability that the random sample contains zero task assignments from $P\%$ of the best-performing assignments. If the number of possible task assignments is large (*i.e.* the population is large), the probability that a single assignment is in the lower $(100 - P)\%$ of the population is $\frac{100-P}{100}$. We assume that the sample is selected from a finite population of all task assignments using sampling with replacement. Sampling with replacement means that at any draw, all assignments in the population are given an equal chance of being drawn, no matter how often they have already been drawn [16]. In addition to this, we assume that the selected task assignments in the sample are mutually independent and uniformly distributed. Taking into account these assumptions, the probability that all $n$ assignments in the sample are contained in the lower $(100 - P)\%$ of the population is computed as: $P(A') = (\frac{100-P}{100})^n$. As $A$ and $A'$ are opposite events, the sum of probabilities that they occur is equal to 1: $P(A) + P(A') = 1$. Therefore, the probability of the event $A$ can be computed as:

$$P(A) = 1 - P(A') = 1 - \left(\frac{\text{100-P}}{100}\right)^{\mathbf{n}}$$

We observe that the probability that a sample of random task assignments contains at least one of the $P\%$ of the best-performing assignment is independent of the population size (*i.e.* the number of possible task assignments). However, we have to be aware that this is valid only for large populations, which is satisfied in the case of task scheduling problems in MMT processors, as we have shown in Table 1.

Figure 2 plots the probability $P(A)$ for the samples of different size and for different percentages of the best-performing task assignments. The $X$ axis of the figure shows the number of assignments in the sample ($n$), while the $Y$ axis presents the probability that the sample contains at least one from $P\%$ of the best-performing task assignments. The figure shows data for $P$ = 1, 2, 5, 10, 25. We derive three conclusions from Figure 2. First, that the probability asymptotically approaches 1 as the number of task assignments in the sample increases. Second, as the fraction of the best-performing assignments decreases (from 25% to 1% in the figure) the probability approaches 1 slower (more task assignments are required to reach a high probability). Finally, we observe

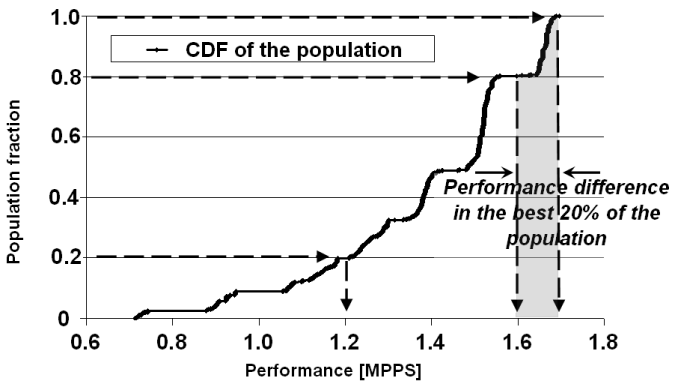



**Figure 3.** An example of Cumulative Distribution Function

that small samples of below 10 elements are unlikely to capture any task assignment from 1%, 2%, and 5% of the best-performing ones. However, a sample of several hundred random observations is sufficient to capture at least one of 1% or 2% of the best-performing task assignments with a very high probability. This means that, if we assume that 1% or 2% of the best-performing assignments have a good performance, simply running several hundred or several thousand randomly selected task assignments is sufficient to capture at least one assignment with a good performance.

### 3.2 Cumulative Distribution Function

One way to determine which portion of the population of all task assignments exhibits a good performance is by plotting the Cumulative Distribution Function (CDF). We illustrate CDF with the following example. Figure 3 shows the CDF of all 1500 task assignments for a network processing a workload of six threads. The details of the experimental framework are described in Section 4. The $X$ axis of the figure shows the measured performance in Millions of Packets processed Per Second (MPPS). The $Y$ axis shows the portion of all task assignments (the population) that exhibit a performance lower or equal to the corresponding value on the $X$ axis.

The presented CDF plot also shows the importance of task assignment. The performance of the task assignments ranges from 0.715 MPPS to 1.7 MPPS, meaning that non-optimal assignment could lead to $\frac{1{,}700{,}000-715{,}000}{1{,}700{,}000} = 58\%$ of performance loss. The performance difference in $P\%$ of the best-performing task assignments can be directly determined from the CDF of the population. For the data presented in Figure 3, the performance difference in 1% of the best-performing task assignments is very low, below 10,000 PPS which is only 0.6% of the optimal system performance. However, in general, since running all task assignments is not feasible, CDF based on the performance of all assignments cannot be constructed. In that case, the measured performance of a sample of task assignments can be used to construct an Empirical CDF (ECDF) that estimates the CDF of the whole population [27, 34]. ECDF is a very good method to estimate the median part of the actual CDF, but it cannot be used to infer the performance of the best task assignments in the tails of CDF of vast populations. Therefore, ECDF cannot be used to estimate (with a hard confidence level) the optimal system performance, nor the performance difference in $P\%$ of the best-performing task assignments, which are the main goals of our study.

### 3.3 Estimation of the optimal performance

One way to evaluate any task assignment approach is to compare the performance of the task assignments provided by the approach with the performance of the best assignment, *i.e.* with the optimal system performance. This performance difference shows the room for possible improvement of the proposed scheduling approach. However, as in general the number of possible task assignments is vast, the optimal system performance cannot be determined [33]. In this paper, we propose using statistical inference methods to estimate the optimal system performance based on the measured performance of a sample of random task assignments.

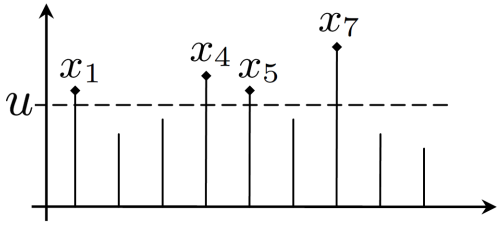



**Figure 4.** Exceedances over the threshold $u$

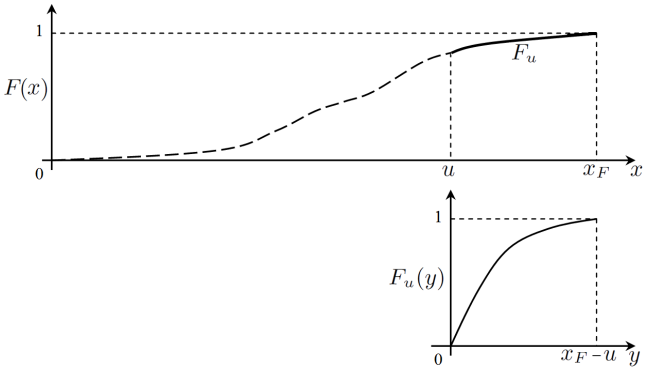



**Figure 5.** Cumulative distribution function $F(x)$ and and corresponding conditional excess distribution function $F_u(y)$

#### 3.3.1 Extreme value theory

We estimate the performance of the best task assignment using Extreme Value Theory (EVT). EVT is a branch of statistics that studies extreme deviations from the median of distributions [10, 11]. One of the EVT approaches is the *Peak Over Threshold* (POT) method. The POT method takes into account the distribution of the observations that exceed a given (high) threshold. For example, in Figure 4, the observations $x_1$, $x_4$, $x_5$, and $x_7$ exceed the threshold *u* and constitute extreme values that can be used in POT analysis.

The POT method can also be explained using cumulative distribution function (CDF). For example, assume that $F$ is the CDF of a random variable $X$. The POT method can be used to estimate the cumulative distribution function $F_u$ of values of $x$ above a certain threshold $u$. The function $F_u$ is called the *conditional excess distribution function* and it is defined as

$$F_u(y) = P(X - u \leq y \mid X > u), \quad 0 \leq y \leq x_F - u,$$

where $X$ is the observed random variable, $u$ is the given threshold, $y = x - u$ are the exceedances over the threshold, and $x_F \leq \infty$ is the right endpoint of the cumulative distribution function $F$. Figure 5 shows a CDF of a random variable $X$ (upper chart) and the corresponding conditional excess distribution function $F_u(y)$ (bottom chart).

The POT method is based on the following theorem [9, 43]:

THEOREM 1. *For a large class of underlying distributions functions $F$, the conditional excess distribution function $F_u(y)$, for $u$ large, is well approximated by $F_u(y) \approx G_{\xi,\sigma}(y)$ where*

$$G_{\xi,\sigma}(y) = \begin{cases} 1-(1+\frac{\xi}{\sigma}y)^{-1/\xi} & for \quad \xi \neq 0 \\ 1-e^{-y/\sigma} & for \quad \xi = 0 \end{cases}$$

*for $y \in [0,(x_F-u)]$ if $\xi \geq 0$ and $y \in [0,-\frac{\sigma}{\xi}]$ if $\xi < 0$, where $G_{\xi,\sigma}$ is called Generalized Pareto Distribution (GPD).*

This means that the $F_u$ of numerous distributions that present real-life problems can be approximated with GPD. For each particular problem, the decision as to whether GPD can be used to model the problem, is made based on how well the sample of observations can be fitted to GPD. We describe the goodness of fit of observations to GPD in Section 3.3.2 in Step 2 and Step 3 of the presented analysis. GPD is defined with two parameters: shape parameter $\xi$ and scaling parameter $\sigma$. One of the characteristics of GPD is that for $\xi < 0$ the upper bound of the observed value (in our study, the performance of the best task assignment) can be computed as $u - \frac{\sigma}{\xi}$, where $\sigma$ and $\xi$ are the GPD parameters and $u$ is the selected threshold [25, 35].

In Theorem 1, the definition of $G_{\xi,\sigma}(y)$ for parameter $\xi = 0$ can only be used to model problems with an infinite upper bound [25, 35]. As in this study we use GPD to estimate application performance when running in a real computer system, the upper bound of the observed value is finite and the estimated values of the parameter $\xi$ are always $\hat{\xi} < 0$. Therefore, for the sake of the simplicity of the presented mathematical formulas, in the rest of the paper we do not present $G_{\xi,\sigma}(y)$ formulas for parameter $\xi = 0$.

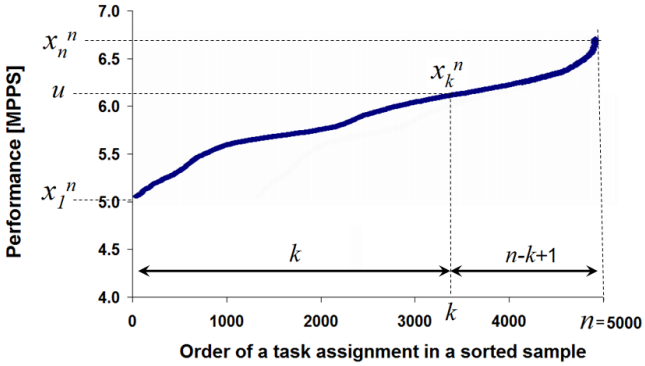
### 3.3.2 Application of Peak Over Threshold method

We use the POT method to estimate the optimal system performance for a given workload (*i.e.* the performance of the best task assignment) based on the measured performance of the sample of random task assignments. The application of the POT method involves the following steps:

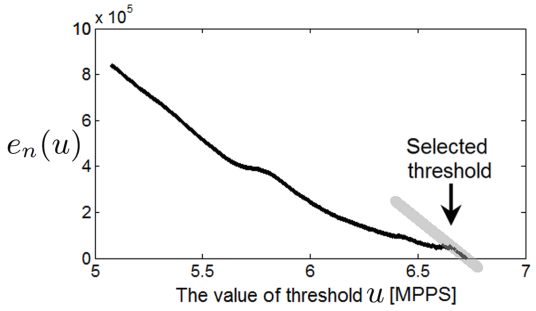
**Step 1:** Generate the sample of random task assignments, execute the assignments on the target machine, and measure the performance of each assignment. A requisite of the presented statistical analysis is that the selected task assignments in the sample are independent and identically distributed (*iid*): they have to be independent and have to have the same marginal distribution. The sample assignments have to be taken from a single population using sampling with replacement [55]. The method we use to generate *iid* task assignments is described next.

For example, assume that a workload of $T$ tasks would be assigned on a processor comprising $V$ hardware contexts, where $T \leq V$. We enumerate the hardware contexts of the processor with integers from 1 to $V$ and for each task in the workload we randomly select an integer from this interval. The number assigned to each running task represents the hardware context to which the task is mapped to in a given task assignment. After mapping all tasks, we check if the generated task assignment is valid. An assignment is not valid if two or more tasks are mapped to the same hardware context. If this is the case, we simply discard the invalid assignment and repeat the whole process until the sample contains the required number of task assignments. As assignments in the sample are independent and they are sampled from a single population using sampling with replacement, the generated sample is comprised of *iid* task assignments.

**Step 2:** Select the threshold $u$. The selection of the threshold $u$ is an important step in POT analysis. Gilli and Këllezi [25, 35] propose using *sample mean excess plot*, a graphical tool for




(a) Ordered sample of task assignments




(b) Sample mean excess plot

**Figure 6.** Selection of the threshold $u$

threshold selection. This method first sorts all task assignments in a sample in non-decreasing performance order: $x_1^n \leq x_2^n \leq \cdots \leq x_n^n$. Figure 6(a) shows the sorted performance of 5000 random task assignments for a workload comprised of 24 threads of the *IPFwd-L1* application (see Section 4). Then, the possible threshold $u$ takes the values from $x_1^n$ to $x_n^n$ ($x_1^n \leq u \leq x_n^n$) and for each value we compute the sample mean excess function $e_n(u)$:

$$e_n(u) = \frac{\sum_{i=k}^{n}(x_i^n - u)}{n-k+1}, \text{ where } k = min\{\ i \mid x_i^n > u\}.$$

In this formula, the factor $n-k+1$ is the number of observations that exceed the threshold. Finally, the sample mean excess plot is defined by the points $(u, e_n(u))$ for $x_1^n \leq u \leq x_n^n$. Figure 6(b) shows the example of the sample mean excess plot for 24 threads of *IPFwd-L1* application.

As we are interested in the upper performance bound estimation, the estimated parameter $\xi$ of GPD has to be negative ($\hat{\xi} < 0$). One of the characteristics of the GPD with parameter $\xi < 0$ is that it has linear mean excess function plot. In order to have a good fit of the conditional distribution function $F_u$ to GPD, the threshold should be selected so that the observations that exceed the threshold have a roughly linear sample mean excess plot. As an example, for the data presented in Figure 6, the threshold should be selected to be around $6.6 \times 10^6$. Sample mean excess plot is also a very good tool to test whether GPD can be used to model a particular set of observations. If the right portion of the mean excess plot for the sample of measured task assignments performance is not (roughly) linear, that particular problem cannot be modeled using GPD.

Another important tool that can be used to understand if a given sample of observations can be modeled with a Generalized Pareto Distribution is a *quantile plot* [10, 35]. In a quantile plot, the sample quantiles $x_i^n$ are plotted against the quantiles of a target distribution $F^{-1}(q_i)$ for $i = 1, \ldots, n$. If the sample data originates from the family of distributions $F$, the plot is close to a straight line. For all experiments presented in this paper, we plot the quantiles of the samples of observations against the quantiles of GPD. In all

experiments, the form of quantile plots strongly suggest that that samples of observations follow a Generalized Pareto Distribution.

The linear sample mean excess plot and the quantile plot are not the only constraints that should be considered when selecting the threshold. If the threshold is too low, the estimated parameters of GPD may be biased to the median values of the cumulative distribution function instead of to the maximum values. In order to avoid this bias, when selecting a threshold we have to ensure that the number of observations that exceed the selected threshold is not higher than 5% of the task assignments in the whole sample. This is a commonly used limit in studies that use POT analysis [25, 35].

**Step 3:** Fit the GPD function to the observations that exceed the threshold and estimate parameters $\xi$ and $\sigma$.

Once the threshold $u$ is selected, the observations over the threshold can be fitted to GPD, and the parameters of the distribution can be estimated. For the sake of simplicity, we assume that observations from $x_k^n$ to $x_n^n$ in the sorted sample presented in Figure 6(a) exceed the threshold. We rename the exceedances $y_{i-k+1} = x_i^n - u$ for $k \leq i \leq n$ and use the set of elements $\{y_1, y_2, ..., y_m\}$ to estimate the parameters of GPD. The number of elements in the set, $m = n - k + 1$, is the number of exceedances over the threshold.

Different methods can be used to estimate the parameters of GPD from a sample of observations [12, 28, 30, 52]. In our study, we used the estimation based on the *likelihood* function. The likelihood is a statistical method that estimates distribution parameters based on a set of observations [8]. The GPD is defined with parameters $\xi$ and $\sigma$. The likelihood that a set of observations $\{y_1, y_2, ..., y_m\}$ is the outcome of a GPD with parameters $\xi = \xi_0$ and $\sigma = \sigma_0$ is equal to the probability that GPD with parameters $\xi_0$ and $\sigma_0$ has the outcome $\{y_1, y_2, ..., y_m\}$.

We will use the likelihood function to compute the probability that different values of GPD parameters have for a given set of observations $\{y_1, y_2, ..., y_m\}$. As the logarithm is a monotonically increasing function, the logarithm of a positive function achieves the maximum value at the same point as the function itself. This means that instead of finding the maximum of a likelihood function, we can determine the maximum of the logarithm of the likelihood function - the *log-likelihood* function. In statistics, log-likelihood is frequently used instead of the likelihood function because it simplifies the computation. The estimation of parameters $\xi$ and $\sigma$ of $G_{\xi,\sigma}(y)$ involves the following steps:

(i) Determine the corresponding probability density function as a partial derivate of $G_{\xi,\sigma}(y)$ with respect to $y$:

$$g_{\xi,\sigma}(y) = \frac{\partial G_{\xi,\sigma}(y)}{\partial y} = \frac{1}{\sigma}(1 + \frac{\xi}{\sigma}y)^{-\frac{1}{\xi}-1}$$

(ii) Find the logarithm of $g_{\xi,\sigma}(y)$:

$$log\ (g_{\xi,\sigma}(y)) = -log\ \sigma - (\frac{1}{\xi} + 1)\ log(1 + \frac{\xi}{\sigma}y)$$

(iii) Compute the log-likelihood function $L(\xi, \sigma|y)$ for the GPD as the logarithm of the joint density of the observations $\{y_1, y_2, ..., y_m\}$:

$$L(\xi, \sigma|y) = \sum_{i=1}^{m} log\ g_{\xi,\sigma}(y_i)$$

$$L(\xi, \sigma|y) = -m\ log\ \sigma - (\frac{1}{\xi} + 1)\sum_{i=1}^{m} log(1 + \frac{\xi}{\sigma}y_i)$$

We compute estimated values of parameters $\hat{\xi}$ and $\hat{\sigma}$, to *maximize* the value of the log-likelihood function $L(\xi, \sigma|y)$ for observations $\{y_1, y_2, ..., y_m\}$:

$$L(\hat{\xi}, \hat{\sigma}|y) = \max_{\xi,\sigma}(L(\xi, \sigma|y))$$

$$L(\hat{\xi}, \hat{\sigma}|y) = \max(-m\ log\ \sigma - (\frac{1}{\xi} + 1)\sum_{i=1}^{m} log(1 + \frac{\xi}{\sigma}y_i))$$

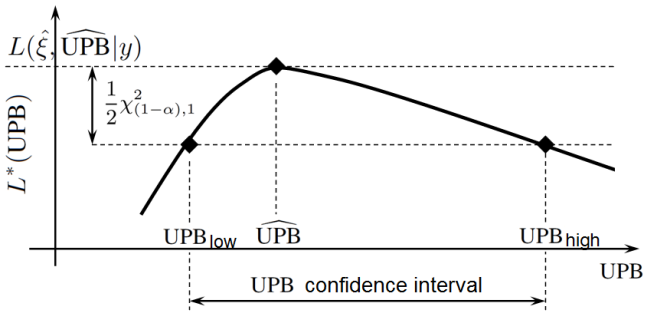



**Figure 7.** UPB confidence interval

In order to determine the parameters $\hat{\xi}$ and $\hat{\sigma}$, we find the minimum of the negative log-likelihood function, $\min_{\xi,\sigma}(-L(\xi, \sigma|y))$, using the procedure *fminsearch()* included in Matlab® R2007a [14]. The values $\hat{\xi}$ and $\hat{\sigma}$ are called the point estimate of the parameters $\xi$ and $\sigma$, respectively.

**Step 4:** Estimate the optimal system performance (the upper performance bound of all task assignments). The upper bound of the observed value can be determined only for $\hat{\xi} < 0$ which is satisfied for all data sets that are presented in this paper. The point estimate of the Upper Performance Bound (UPB) is computed as $\widehat{\text{UPB}} = u - \hat{\sigma}/\hat{\xi}$.

In order to indicate the confidence of the estimate, we compute the confidence intervals of the estimated $\widehat{\text{UPB}}$. UPB confidence interval is computed using *likelihood ratio test* [8] which consists of the following steps:

(i) Define GPD as a function of $\xi$ and UPB:

$$G_{\xi,\text{UPB}}(y) = 1 - (1 - \frac{1}{\text{UPB} - u}y)^{-1/\xi}$$

(ii) Determine the corresponding probability density function:

$$g_{\xi,\text{UPB}}(y) = \frac{\partial G_{\xi,\text{UPB}}(y)}{\partial y} = -\frac{1}{\xi(\text{UPB} - u)}(1 - \frac{1}{\text{UPB} - u}y)^{-\frac{1}{\xi}-1}$$

(iii) Compute the joint log-likelihood function for observations $\{y_1, ..., y_m\}$:

$$L(\xi, \text{UPB}|y) = \sum_{i=1}^{m} log\ g_{\xi,\text{UPB}}(y_i)$$

$$L(\xi, \text{UPB}|y) = -n\ log(-\xi(\text{UPB} - u)) - (1 + \frac{1}{\xi})\sum_{i=1}^{n} log(1 - \frac{1}{\text{UPB} - u}y_i)$$

(iv) Find the UPB confidence interval. We determine the confidence interval for UPB using likelihood ratio test [8] and Wilks's theorem [15, 59, 60]. The maximum log-likelihood function is determined as $L(\hat{\xi}, \widehat{\text{UPB}}|y) = \max_{\xi,\text{UPB}}(L(\xi, \text{UPB}))$.

The function $L(\hat{\xi}, \widehat{\text{UPB}}|y)$ has two parameters that are free to vary ($\xi$ and UPB), hence it has two degrees of freedom, $df_1 = 2$. As UPB is our parameter of interest, the profile log-likelihood function is defined as $L^*(\text{UPB}) = \max_{\xi} L(\xi, \text{UPB})$.

The function $L^*(\text{UPB})$ has one parameter that is free to vary *i.e.* one degree of freedom, $df_2 = 1$. Wilks's theorem applied to the problem that we are addressing claims that, for large number of exceedances over the threshold, distribution of $2(L(\hat{\xi}, \widehat{\text{UPB}} - L^*(\text{UPB}))$ converges to a $\chi^2$ distribution with $df_1 - df_2$ degrees of freedom. Therefore, the confidence interval of UPB includes all values of UPB that satisfy the following condition:

$$L(\hat{\xi}, \widehat{\text{UPB}}) - L^*(\text{UPB}) < \frac{1}{2}\chi^2_{(1-\alpha),1} \quad (1)$$

$\chi^2_{(1-\alpha),1}$ is the $(1 - \alpha)$-level quantile of the $\chi^2$ distribution with one degree of freedom ($df_1 - df_2 = 1$). $\alpha$ is the confidence level for which we compute UPB confidence intervals.

We illustrate the computation of the UPB confidence interval in Figure 7. The figure plots $L^*(\text{UPB})$ for different values of UPB. For UPB $= \widehat{\text{UPB}}$, $L^*$ reaches its maximum. The confidence interval of UPB includes all values of UPB that satisfy the condition $L^*(\text{UPB}) > L(\hat{\xi}, \widehat{\text{UPB}}) - \frac{1}{2}\chi^2_{(1-\alpha),1}$ which corresponds to the Equation 1. We computed the UPB confidence interval using an iterative method based on the *fminsearch()* function included in Matlab® R2007a.

The code that generates the sample mean excess plots, infers the parameters of the GPD distribution, and estimates the optimal system performance was developed in Matlab® R2007a.

### 3.4 Summary

In this section, we have presented two statistical methods. The first method computes the probability that a sample of $n$ randomly selected task assignments captures at least one out of $P\%$ (*e.g.* $1\%$) best performing assignments. The results of this method show that running several hundred or several thousand random tasks assignments is enough to capture at least one out of $1\%$ of the best performing assignments with a very high probability. The second method estimates the performance of the best-performing task assignment, *i.e.* the optimal system performance for a given workload. The method infers the optimal system performance with the hard statistical confidence based on a measured performance of the sample of random task assignments. The presented method is completely independent of the hardware environment and target applications. The method scales to any number of cores and hardware contexts per core and it does not require any profiling of the application nor does it require knowledge of the architecture of the target hardware.

## 4. Experimental environment

We evaluated the presented statistical analysis for the case study of multithreaded network applications running in a real industrial environment. The environment comprised two T5220 machines that managed the generation and the processing of network traffic. Each T5220 machine comprised one UltraSPARC T2 processor. One T5220 machine executed the Network Traffic Generator (NTGen) [4]. NTGen is a software tool, developed by Oracle that generates IPv4 TCP/UDP packets with configurable options to modify various packet header fields. NTGen transmitted network packets through a 10Gb link to the second T5220 machine in which we executed the task assignments selected by our method. In all the experiments presented in the study, NTGen generated enough traffic to saturate the network processing machine. Thus, in all experiments, the performance bottleneck was the speed at which the packets were processed, which is determined by the performance of the selected task assignment.

### 4.1 UltraSPARC T2 processor

The UltraSPARC T2 is a multithreaded processor [1][2] that comprises eight cores connected through the crossbar to the shared L2 cache (see Figure 8). Each of the cores supports eight hardware contexts, thus up to 64 tasks can be simultaneously executed on the processor. Strands inside each hardware core are divided into two groups of four strands, forming two hardware execution pipelines. Therefore, tasks simultaneously running on the UltraSPARC T2 processor can share (and compete for) different resources in three different levels depending on how they are distributed on the processor. Resources at the *IntraPipe* level, such as the Instruction Fetch Unit (IFU) and the Integer Execution Units (IEU) are shared among tasks running in the same hardware pipeline. The *IntraCore* resources, such as the L1 instruction cache, L1 data cache, instruction and data TLBs, Load Store Unit (LSU), Floating Point and Graphic Unit (FPU), and Cryptographic Processing Unit are shared among tasks running on the same core. Finally, the resources shared among all tasks simultaneously running on the processor (*InterCore* sharing level) are mainly the L2 cache, the on-chip interconnection network (crossbar), the memory controllers, and the interface to off-chip resources [56].

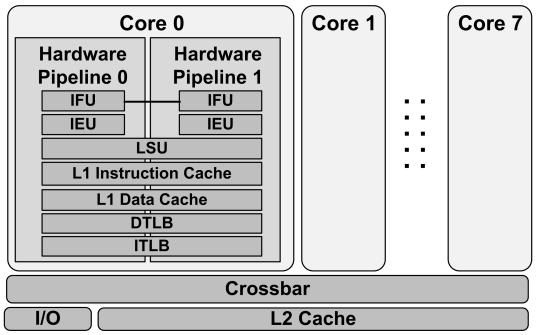



**Figure 8.** Schematic view of the three resource sharing levels of the UltraSPARC T2 processor

### 4.2 Netra DPS

Networking systems use lightweight runtime environments to reduce the overhead introduced by fully-fledged OSs. One of these environments is Netra DPS [3, 4] developed by Oracle. Netra DPS does not incorporate virtual memory nor a run-time process scheduler, and performs no context switching. The assignment of running tasks to processor hardware contexts (virtual CPUs) is performed statically at the compile time. It is the responsibility of the programmer to define the hardware context in which each particular task will be executed. Netra DPS does not provide any interrupt handler nor daemons. A given task runs to completion on the assigned hardware context without any interruption.

### 4.3 Benchmarks

Netra DPS is a specific lightweight runtime environment that does not provide functionalities of fully-fledged OSs such as system calls, dynamic memory allocation, or file management. Therefore, benchmarks included in standard benchmark suites have to be adopted in order to execute in this environment. The benchmarks used are described next:

**(1) IP Forwarding (IPFwd)** is one of the most representative Layer2/Layer3 network applications. IPFwd application makes the decision to forward a packet to the next hop based on the destination IP address. Depending on the size of the lookup table and destination IP addresses of the packets that are to be processed, the IPFwd application may have significantly different memory behavior. In order to cover different cases of IPFwd memory behavior, we created two variants of the IPFwd application that are based on the IPFwd application included in the Netra DPS distribution [3]:

(i) The lookup table fits in the L1 data cache (**IPFwd-L1**);

(ii) The lookup table entries are initialized to make IPFwd continuously access the main memory (**IPFwd-Mem** benchmark).

IPFwd-L1 is representative of the best case of IPFwd memory behavior, since it shows high locality in data cache accesses. On the other hand, IPFwd-Mem represents the worst case of IPFwd memory behavior used in network processing studies, in which there is no cache locality between accesses to the lookup table [47].

**(2) Packet analyzer** is a program that can intercept and log traffic passing over a network or part of a network [17]. Packet analyz-

ers are primary tolls for network monitoring and management used to troubleshoot network problems, examine security issues, gather and report network statistics, detect suspect content, and filter it from the network traffic [51, 54, 61].

The packet analyzer that we used in the experiments captures each packet that passes through the Network Interface Unit (NIU), decodes the packet, and analyzes its content according to the appropriate RFC specifications [31]. The packet analyzer can display the information about different fields of packet headers at Layer 2, Layer 3, and Layer 4, and about the packet payload. A user can decide to log all traffic that passes through the NIU, or to define filters based on many criteria. In the experiments presented in this paper, we used the packet analyzer to log MAC source and destination address, time to live field, Layer 3 protocol, source and destination IP address, and source and destination port number of all packets passing through the NIU of the processor under study.

**(3) Aho-Corasick** is a string matching algorithm. String matching is the basic technique to analyze the network traffic at the application layer [29]. In networking, string matching algorithms search for a set of strings (keywords) in the payload of the network packets. Aho-Corasick is an efficient algorithm that locates all occurrences of a given set of keywords in a string of text (packet payload in case of packet processing). The algorithm constructs a finite state pattern matching machine from the keywords and then uses the pattern matching machine (finite automata) to process the string of text in a single pass [7]. The Aho-Corasick string matching algorithm has proven linear performance, and it is suitable for searching of a large set of keywords concurrently. This is why the Aho-Corasick algorithm is used in state-of-the art network intrusion detection systems such as Snort [40].

In the experiments presented in the paper, we used the Aho-Corasick algorithm to search for keywords from Snort Denial-of-Service set of intrusion detection rules (version 2.9, November 2011) in the payloads of the packets that were processed.

**(4) Stateful packet processing** is an important component of state-of-the art network monitoring tools [41, 58] and intrusion prevention and detection system [51]. Unlike stateless applications that process each packet independently (like the IPFwd, Packet analyzer, and Aho-Corasick benchmarks used in this study), stateful packet processing keeps the information of previous packet processing.

The packets that belong to the same *flow*, *i.e.* have the same *flow-keys* [3], share the common information called the *flow-record* [19]. The record of a given flow contains the information as to whether the flow is open (the connection is established), safe, malicious, *etc.* The information about the active flows is stored in a hash table that is indexed based on the flow-keys. The common main components of stateful packet processing are: (1) Read the flow-keys of a packet; (2) Use a hash function to determine the corresponding hash table entry based on the packet flow-keys; (3) Access the hash table. Lock, read, and update the flow-record of an already-existing flow, or create a flow-record for a new flow.

The stateful packet processing benchmark that we used in the experiments is comprised of these three components. The stateful benchmark uses the same hash function as the one that is implemented in *nProbe* network monitor [19, 41]. The hash table contains $2^{16}$ entries, which is sufficient to store the records of active flows of fully-utilized 10Gb link [57], and it is the hash table size that was already used to test different network monitoring tools [19].

[3] The flow-keys are typically the source and destination IP address, the source and destination port, and protocol used.

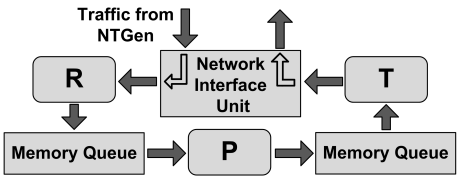



**Figure 9.** The schematic view of the benchmarks

#### 4.3.1 Benchmark implementation

Each benchmark is divided into three threads that form a software pipeline (see Figure 9). This is a commonly-used approach in the development of the network applications [4, 62]:

- The receiving threads (R) of all benchmarks read the packets from the Network Interface Unit (NIU) associated with the receiver 10Gb network link, and write the pointers to the packets into the R→P memory queues.
- The processing threads (P) read the pointer to the packets from the memory queues, process the packets, and write the pointers to the P→T memory queues. The packet processing is different for each benchmark, *e.g.* P threads of the *IPFwd-L1* and *IPFwd-Mem* benchmarks read the destination IP address, call the hash function, and access the lookup table; P threads of the *Aho-Corasick* benchmark search for the keywords in the packet payload, *etc.*
- Finally, the transmitting threads (T) of all benchmarks read the pointers from the P→T memory queues, and send the packets to the network through the NIU associated to the 10Gb network link.

To summarize, the presented benchmarks represent a good testbed for the analysis of tasks assignment techniques because:

(1) Each benchmark is divided in three different threads, thus the systems deal with heterogeneous tasks even when several instances of the same application are executed simultaneously.

(2) The benchmarks stress the hardware resources of the UltraSPARC T2 processor at all three sharing levels [56].

(3) Each instance of the benchmarks comprises interconnected threads that communicate through shared memory queues. The performance of the benchmarks also depend on the distribution of interconnected threads among processor cores (L1 cache domains).

(4) The impact of task assignment to the performance is significant. We detect performance variation of up to 49% between different task assignments of the same workload.

### 4.4 Methodology

In order to assure stable results, we measured the performance of task assignments when each application instance processed three million network packets. This means that each application thread was executed three million times. The execution time of each experiment was around 1.5 seconds, and the duration depended on the benchmark and on the distribution of the simultaneously running threads.

## 5. Results

In this section, we apply the presented statistical approach to the task assignment of multithreaded network applications running on the UltraSPARC T2 processor. We start by analyzing whether a sample of random task assignments can capture an assignment with a good performance. Next, we estimate the optimal system performance for a given workload, and compare it with the measured performance of the observed best task assignment in the sample.

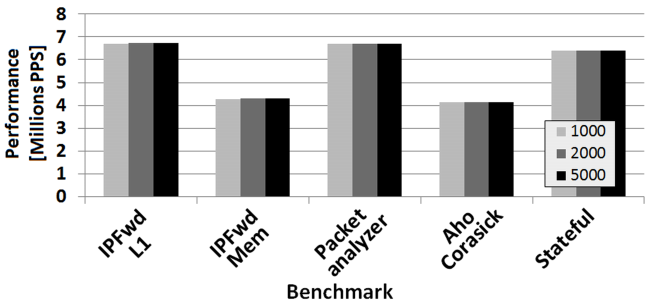



**Figure 10.** Performance of the best task assignment in the random sample

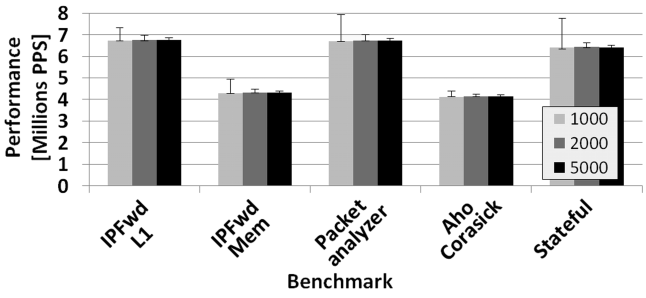



**Figure 11.** Estimated best-performing schedule performance

Finally, we show how the presented analysis can be applied in the industrial case study for task assignment of network applications.

In all presented experiments, we simultaneously executed eight benchmark instances (24 threads). We could not execute more than eight benchmark instances because of the limitation in the experimental environment: the on-chip Network Interface Unit (NIU) of the UltraSPARC T2 used in the study can split the incoming network traffic into up to eight DMA channels and Netra DPS binds at most one receiving thread to each DMA channel. As a part of future work, we plan to apply the presented statistical approach to applications with several processing threads and to workloads with a higher number of simultaneously-running tasks.

### 5.1 Finding task assignments with good performance

In Section 3.1, we presented a method to compute the probability that a sample of randomly selected task assignments contains at least one of $P\%$ of best-performing assignments. We showed that samples containing more than several hundred random task assignments capture at least one in $1\%$ or $2\%$ of the best performing task assignments with a probability higher than $99\%$. We also showed that the probability that a sample of random assignments contains at least one out of $1\%$ of the best-performing task assingments asymptotically approaches 1 as the number of task assignments in the sample exceeds 1000 (see Figure 2).

In order to analyze if further increasing the number of task assignments in the sample increases the performance of the captured best-performing assignment, we executed experiments for 1000, 2000, and 5000 random task assignments. The results of the experiment are plotted in Figure 10, in which the $X$ axis lists different benchmarks and the $Y$ axis shows the performance of the best task assignment in the sample, measured in processed Packets Per Second (PPS). We observed that increasing the sample size from 1000 to 5000 only negligibly improves the performance of the captured best task assignment. The highest performance improvement we detect is only $0.6\%$ for the *IPFwd-Mem* benchmark. For the remaining four benchmarks, the performance improvement when the sample increases from 1000 to 5000 assignments is below $0.25\%$.

We repeated the same set of experiments for different workloads each having a different number of simultaneously running threads. The conclusions that we reached are the same - increasing sample size from 1000 to 5000 insignificantly improves the performance of the captured best-performing task assignment.

In order to analyze how good the performance of the captured best task assignment in the sample is, and what its performance loss with respect to the optimal one is, we used the statistical inference method that estimates the performance of the optimal task assignment.

### 5.2 Estimation of the optimal performance

In Section 3.3, we described the statistical approach for the estimation of the optimal system performance for a given workload. The presented statistical method estimates the performance of the best task assignment based on the measured performance of a sample of random assignments. Figure 11 shows the estimated optimal system performance for all five benchmarks in the suite which are listed along $X$ axis. In order to understand the impact of sample size to the estimated optimal system performance, we executed experiments and present data for 1000, 2000, and 5000 random task assignments in the sample. The height of the solid bar corresponds to the point estimation of the optimal system performance, while the error bars show the confidence interval for the 0.95 confidence level.

From the data presented in Figure 11, we can see that the point estimation is roughly the same for all three sample sizes. On the other hand, we see that for four out of five benchmarks (all except *Aho-Corasick*) increasing the sample size significantly narrows the confidence interval, *i.e.* it improves the precision of the estimation. As we increase the number of assignments in the sample, more task assignments in the right tail of the cumulative distribution function (see Figure 5) are used to estimate the parameters of the Generalized Pareto Distribution (GPD). As we stated in Section 3, in order to avoid the bias of the GPD to median values in the cumulative distribution function, no more than $5\%$ of the best performing assignments should be considered when the right tail of the cumulative distribution function is fitted to GPD. Therefore, the maximum number of observations we use to estimate a performance of the optimal system performance is 50, 100, and 250 for the sample of 1000, 2000, and 5000 task assignments, respectively. As we increase the number of task assignments in the sample, more observations can be used to estimate the optimal system performance, which, in turn, leads to more precise estimations.

Figure 12 shows the performance difference between the best-performing assignment in the sample and the estimated optimal system performance. This chart shows how good the performance of the captured best assignment in the given sample is. The benchmarks that are used in the case study are listed along the $X$ axis. Different bars present results for 1000, 2000, and 5000 random assignments in the sample. The height of the solid bar corresponds to the point estimation of the optimal system performance, while the error bars correspond to the confidence interval for the 0.95 confidence level. We reach several conclusions from the results that are presented in Figure 12. For 1000 task assignments in the sample, the estimated possible performance improvement is considerable and significantly different for different benchmarks. For *Aho-Corasick* and *IPFwd-L1* benchmarks, the detected possible performance improvement ranges up to $7\%$ and $9\%$. For *IPFwd-Mem, Packet analyzer,* and *Stateful* benchmarks, the possible performance improvement ranges up to $16\%$, $19\%$, and $23\%$, respec-

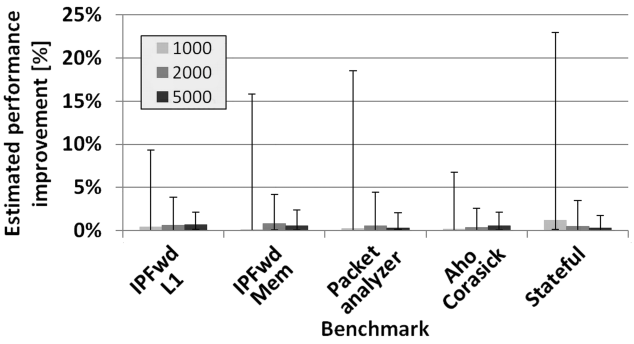



**Figure 12.** Estimated performance improvement

tively. For 2000 assignments in the sample, the performance difference between the best assignment in the sample and the estimated optimal performance is below 5% for all five benchmarks in the suite. Finally, for the 5000 task assignments in the sample, the best measured performance in the random sample is very close to the estimated optimal performance, for all five benchmarks in the suite. The highest possible performance improvement is only 2.4% (*IPFwd-Mem* benchmark).

Overall, in this section, we applied the method presented in Section 3 to estimate the optimal system performance for a given workload. We used the estimated performance to understand how good the best task assignments captured in the random samples are. In our experiments, running only several thousand random task assignments from a vast population was enough to detect assignments with performance very close to the optimal ones. We also showed that increasing the sample size from 1000 to 5000 significantly improved the precision of the estimation, though it only insignificantly improved the performance of the captured best task assignment.

### 5.3 Case study

Next, we show how the presented analysis can be applied to an industrial case study for the task assignment of network applications. In the presented scenario, the main objective is to satisfy the given performance requirement of the best task assignment that is captured in the random sample. In this scenario, the customer requires that the performance difference between the captured best task assignment and the optimal system performance for the given workload is below $X\%$.

In order to address this problem, we developed an iterative algorithm that converges to the performance required by the customer by increasing the number of random assignments in the sample. The schematic view of the algorithm is shown in Figure 13. The algorithm is comprised of four steps.

In **Step 1**, we select the initial sample size $N_{init}$, generate a sample of $N_{init}$ random task assignments, execute them on the target processor, and measure the performance of each assignment in the sample. The output of Step 1 is the measured performance of all task assignments that are executed on the target processor.

In **Step 2**, we apply the described statistical method to estimate the optimal system performance. There are two outputs from Step 2: the performance of the best assignment in the sample and the estimated optimal system performance, *i.e.* the performance of the optimal assignment for a given workload.

In **Step 3**, we compute the performance difference between the observed best assignment in the random sample and the estimated optimal system performance. If this performance difference is ac-

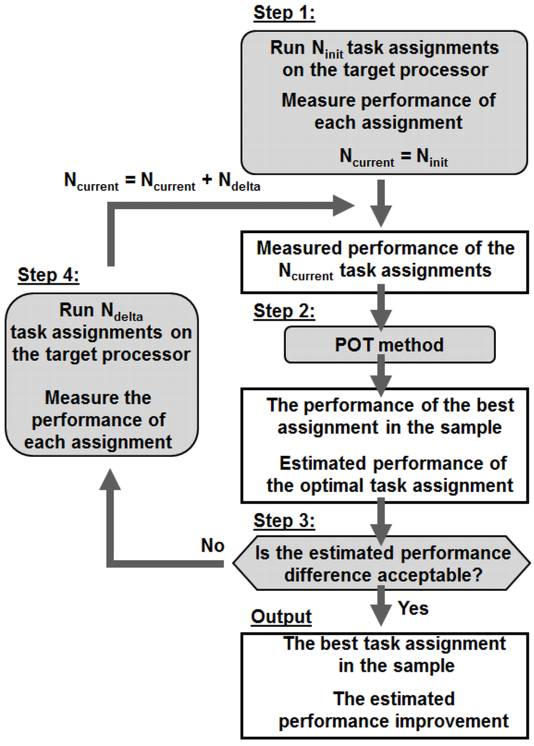



**Figure 13.** Case study: The schematic view of the algorithm

ceptable for the customer (it is below $X\%$), then the iterative process ends. The final outcome of this process is the observed best assignment in the sample and the estimated performance difference with respect to the optimal assignment. On the other hand, if the difference between the performance of the best task assignment in the sample and the estimated optimal system performance is not acceptable (it is higher than $X\%$), the iterative process continues.

In **Step 4**, we generate a sample of $N_{delta}$ random assignments, execute the assignments on the target processor, and measure the performance of each of them. The set of $N_{delta}$ measured values is included in the set of measurements that are the input to the statistical analysis in Step 2 ($N_{current} = N_{current} + N_{delta}$) and the statistical analysis is repeated, this time for a larger input dataset. As the number of task assignments in the random sample increases, the performance of the captured best assignment increases as well. But, more importantly, the input dataset for statistical analysis increases, which provides a more precise estimation of optimal system performance.

**Step 2**, **Step 3**, and **Step 4** of the algorithm are repeated as long the best task assignment in the sample does not satisfy performance requirements specified by the customer.

We appled the presented algorithm to the set of network benchmarks executing in Netra DPS low-overhead runtime environment on the UltraSPARC T2 processor. We started the algorithm with $N_{init} = 1000$ task assignments and in each iteration we executed $N_{delta} = 100$ assignments more. We analyzed three cases, when the acceptable performance loss is 2.5%, 5%, and 10%. In Step 2 of the algorithm, the optimal system performance was estimated for the 0.95 confidence level. The number of task assignments in the sample needed to capture an assignment with the acceptable performance loss is presented in Figure 14. We present data for all five benchmarks in the suite that are listed along the $X$ axis. From the results presented in the figure we see that: (1) Running several

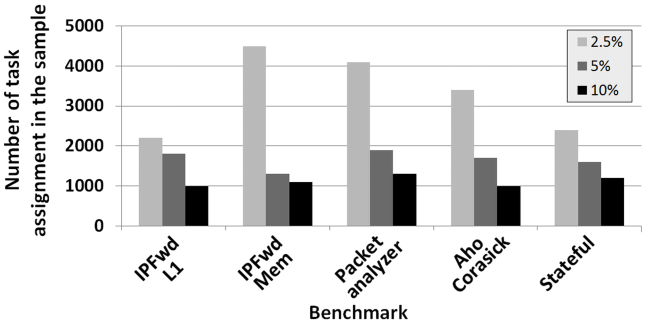



**Figure 14.** Case study: Required number of task assignments

thousand random task assignments was enough to capture an assignment with a performance loss of below 2.5% with respect to the estimated optimal performance. The required number of task assignments range from 2200 for *IPFwd-L1* to 4500 for *IPFwd-Mem* benchmark. (2) As the acceptable performance loss increased, less assignments in the sample were needed. For example, when the acceptable performance loss was 10%, running less then 1300 random task assignments was sufficient to provide the required performance for all five benchmarks. (3) Required number of tasks assignments in the sample depends on the concrete benchmark.

One of the main strengths of the presented approach is that it is application and architecture independent, and that it can be applied to find task assignments that satisfy different performance requirements. The number of assignments in the sample depends on the characteristics of the benchmarks, characteristics of the target architecture, and also on the performance requirements.

### 5.4 Other Considerations

There are a couple of aspects to consider regarding the presented approach.

**Experimental time**: Our method requires execution on the target architecture of all task assignments in the sample. In our target networking environment, as described in Section 4.4, around 1.5 seconds were enough to take a stable measurements of the performance of each task assignment. Hence, the time needed to execute all experiments for samples of 1000, 2000, and 5000 task assignments, for which we have obtained close-to-the-optimal performance, was approximately 25 minutes, 50 minutes, and 2 hours, respectively. This experimentation time is reasonable considering that the selected task assignment can be used during the lifetime of the system.

However, it may be the case that in some environments the time required to execute thousands of experiments on the target architecture is large or unfeasible. In that case, instead or execution of random task assignments on a target processor, the performance of each assignment in the sample can be predicted using a performance predictor. For that matter, the input data to the statistical model is the predicted performance for a sample of task assignments. Performance predictors are models that estimate the performance of task assignments based on the analysis of the target architecture and resource requirements of each task in a workload (see Section 6). It is important to note that the accuracy of the integrated approach that combines performance predictors and statistical analysis depends on the accuracy of the predictor that is used. Design and evaluation of such an integrated approach is part of our future work.

**Workload selection**: As we mentioned in Section 1, for processors with one level of resource sharing, the task scheduling is done in a single step, called *workload selection*: out of all ready-to-run tasks, the OS selects a set of tasks (workload) that will concurrently execute on the processor [32]. As all tasks share the same processor resources homogeneously, the way they interfere is independent of their distribution. In processors with several levels of resource sharing, task scheduling requires an additional step, called *task assignment*. Once the workload is selected, the tasks have to be distributed among different hardware context of the processor.

In the industrial networking environment used in our experimental setup, the workload is known beforehand and cannot be changed at runtime (see Section 4). Hence, in this kind of environment, the workload selection is not an issue and the optimal task assignment is the only scheduling problem. In this paper, we have shown how our statistical approach can be applied to the problem of the optimal task assignment. In processors with one level of resource sharing, the presented methodology can be directly applied to address the workload selection problem. The designer has to generate a sample of random workloads, run them on the target machine, measure the performance of each workload, and follow the methodology we presented in Section 3.

The application of the presented statistical approach becomes more complex for processors with several levels of resource sharing. In this case, the processor may execute different workloads, and each workload may have a different performance for different task assignments. The development of a statistical approach that addresses a combined *workload selection and task assignment* problem is part of our future work.

## 6. Related work

**Workload Selection**: Several approaches that address the workload selection problem propose models that predict the impact of interferences among co-running tasks to system performance. Snavely et al. [49, 50] present the SOS scheduler, which is, to the best of our knowledge, the first scheduler that uses profile-based information to compose workloads. The SOS scheduler uses hardware performance counters to find schedules that exhibit good performance. Eyerman and Eeckhout [22] propose probabilistic job symbiosis model that enhances the SOS scheduler. Based on the cycle accounting architecture [21], the model estimates the single-threaded progress for each job in a multithreaded workload. Other approaches [13, 23, 36, 45] propose techniques to construct workloads of tasks that exhibit good symbiosis in shared caches solving problems of cache contention.

**Task Assignment**: Several studies show that the performance of applications running on multithreaded processors depends on the interference in hardware resources, which, in turn, depends on task assignment [6, 20, 44]. Acosta et al. [6] propose a task assignment algorithm for CMP+SMT processors that takes into account not only the workload characteristics, but also the underlying instruction fetch policy. El-Moursy el al. [20] also focus on CMP+SMT processors and propose an algorithm that uses hardware performance counters to profile task behavior and assign compatible tasks on the same SMT core. Radojković et al. [44] present a model for task assignment of applications running on multithreaded processors. The solution can be used when the number of tasks is low, and it requires profiling information from the application and knowledge about the target architecture.

Other studies analyze task scheduling for platforms comprised of several multithreaded processors [39, 53]. McGregor et al. [39] introduce new scheduling policies that use run-time information from hardware performance counters to identify the best mix of tasks to run across processors and within each processor. Tam el al. [53] present a run-time technique for the detection of data sharing among different tasks. The proposed technique can be used

by an operating system job scheduler to assign tasks that share data to the same memory domain (same chip or the same core on the chip).

Kumar et al. [38] and Shelepov et al. [46] propose algorithms for scheduling in *heterogeneous* multicore architectures. The focus of these studies is to find an algorithm that matches the application's hardware requirements with the processor core characteristics. In our study, we explore interference among tasks that are distributed among the *homogeneous* hardware domains (processor cores and hardware pipelines) of a processor.

Other studies propose solutions for optimal assignment of multithreaded network workloads in parallel processors, specifically in network processors. Kokku et al. [37] propose an algorithm that assigns network processing tasks to processor cores with the goal of reducing the power consumption. Wolf et al. [62] propose run-time support that considers the partitioning of applications across processor cores. The authors address the problem of dynamic threads re-allocation because of network traffic variations, and provide task assignment solutions based on the application profiling and traffic analysis.

**Optimal performance analysis**: To the best of our knowledge, the work of Jiang et al. [33] is the only systematic study devoted to find the optimal task assignment of applications running on multithreaded processors. First, the authors analyze the complexity of the task assignment for multithreaded processors. Later, they propose several task assignment algorithms. The authors use graphs to model interaction between simultaneously-running tasks and use graph search to find the optimal solution. The main drawback of this study is that it assumes that the impact of task interaction to system performance is known beforehand for all possible assignments.

We present a different approach for finding the performance of the optimal task assignment. We do not try to find the best-performing assignment, but to capture a task assignment with performance close to the optimal one. In our approach, the optimal system performance is estimated using statistical inference based on measured performance of a sample of random task assignments.

## 7. Conclusions

Optimal task assignment is one of the most promising ways to improve the performance of applications running on massively multithreaded processors. However, finding an optimal task assignment on modern multithreaded processors is an NP-complete problem.

In this paper, we proposed a statistical approach to the problem of optimal task assignment. In particular, we showed that running a sample of several hundred or several thousand random task assignments is enough to capture at least one out of 1% of the best-performing assignments with a very high probability. We also described the method that estimates, with a given confidence level, the optimal system performance for given workload. Knowing the optimal system performance improves the evaluation of any task assignment technique and it is the most important piece of information for the system designer when deciding whether any scheduling algorithm should be enhanced.

The presented approach is completely independent of the hardware environment and target applications. The approach scales to any number of cores and hardware contexts per core and it does not require any profiling of the application nor does it require knowledge of the architecture of the target hardware.

We successfully applied our proposal to a case study of task assignment of multithreaded network applications running on the UltraSPARC T2 processor. Our results showed that running several thousand random task assignments provided enough information for the precise estimation of the performance of the optimal task assignment, and that it was sufficient to capture the assignments with performance very close to the optimal ones (less than 2.5% of the performance loss), requiring around two hours of experimentation in the target architecture in the worst case.

## Acknowledgments

This work has been supported by the Ministry of Science and Technology of Spain under contract TIN-2007-60625. Petar Radojković and Vladimir Čakarević hold the FPU grant (Programa Nacional de Formación de Profesorado Universitario) under contracts AP2008-02370 and AP2008-02371, respectively, of the Ministry of Education of Spain. This work has been partially supported by the Department of Universities, Research and Information Society (DURSI) of the Catalan Government (grant 2010-BE-00352). The authors wish to thank to Liliana Cucu-Grosjean and Luca Santinelli from INRIA, and Jochen Behrens and Aron Silverton from Oracle for their technical support.

## References

[1] *OpenSPARC$^{TM}$ T2 Core Microarchitecture Specification*. Sun Microsystems, Inc, 2007.

[2] *OpenSPARC$^{TM}$ T2 System-On-Chip (SOC) Microarchitecture Specification*. Sun Microsystems, Inc, 2007.

[3] *Netra Data Plane Software Suite 2.0 Update 2 Reference Manual*. Sun Microsystems, Inc, 2008.

[4] *Netra Data Plane Software Suite 2.0 Update 2 User's Guide*. Sun Microsystems, Inc, 2008.

[5] *Software. Hardware. Complete.* http://www.oracle.com/ocom/groups /public/@ocom/documents /webcontent/044518.pdf, 2010.

[6] C. Acosta, F. Cazorla, A. Ramirez, and M. Valero. Thread to Core Assignment in SMT On-Chip Multiprocessors. In *SBAC-PAD '09: Proceedings of the 21st International Symposium on Computer Architecture and High Performance Computing*, 2009.

[7] A. V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM*, 18, 1975.

[8] A. Azzalini. *Statistical Inference Based on the Likelihood*. Chapman and Hall, London, 1996.

[9] A. A. Balkema and L. de Haan. Residual life time at great age. *Annals of Probability*, 2:792–804, 1974.

[10] J. Beirlant, Y. Goegebeur, J. Segers, and J. Teugels. *Statistics of Extremes: Theory and Applications*. John Wiley and Sons, Ltd, 2004.

[11] E. Castillo. *Extreme value theory in engineering*. Academic Press, Inc., 1988.

[12] E. Castillo and A. Hadi. Fitting the Generalized Pareto Distribution to data. *Journal of the American Statistical Association*, 92, 1997.

[13] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, 2005.

[14] S. J. Chapman. *Essentials of MATLAB Programming*. Cengage Learning, 2009.

[15] H. Chernoff. On the distribution of the likelihood ratio. *Annals of Mathematical Statistics*, 25, 1954.

[16] W. G. Cochran. *Sampling techniques*. Wiley, 1977.

[17] K. J. Connolly. *Law of Internet Security and Privacy*. Aspen Publishers, 2003.

[18] M. De Vuyst, R. Kumar, and D. Tullsen. Exploiting unbalanced thread scheduling for energy and performance on a CMP of SMT processors. In *IPDPS '06: Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium*, 2006.

[19] D. Eckhoff, T. Limmer, and F. Dressler. Hash Tables for Efficient Flow Monitoring: Vulnerabilities and Countermeasures. In *LCN '09: Proceedings of the 34th Conference on Local Computer Networks*, 2009.

[20] A. El-Moursy, R. Garg, D. Albonesi, and S. Dwarkadas. Compatible phase co-scheduling on a CMP of multi-threaded processors. In *IPDPS '06: Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium*, 2006.

[21] S. Eyerman and L. Eeckhout. Per-thread cycle accounting in SMT processors. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, 2009.

[22] S. Eyerman and L. Eeckhout. Probabilistic job symbiosis modeling for SMT processor scheduling. In *ASPLOS '10: Proceeding of the 15th international conference on Architectural support for programming languages and operating systems*, 2010.

[23] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, 2005.

[24] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[25] M. Gilli and E. Këllezi. An application of extreme value theory for measuring financial risk. *Computational Economics*, 27, 2006.

[26] R. Gioiosa, F. Petrini, K. Davis, and F. Lebaillif-Delamare. Analysis of system overhead on parallel computers. In *ISSPIT '04: 4th IEEE International Symposium on Signal Processing and Information Technology*, Rome, Italy, 2004.

[27] M. Greenwood. The natural duration of cancer. *Reports on Public Health and Medical Subjects*, 33:1–26, 1926.

[28] S. Grimshaw. Computing the maximum likelihood estimates for the Generalized Pareto Distribution to data. *Technometrics*, 35, 1993.

[29] F. Guo and T. Chiueh. Traffic Analysis: From Stateful Firewall to Network Intrusion Detection System. In *RPE Report*, 2004.

[30] J. R. M. Hosking and J. R. Wallis. Parameter and quantile estimation for the generalised pareto distribution. *Technometrics*, 29, 1987.

[31] Internet Engineering Task Force (IETF). Request for Comments (RFC), http://www.rfc-editor.org/.

[32] R. Jain, C. J. Hughes, and S. V. Adve. Soft real- time scheduling on simultaneous multithreaded processors. In *RTSS '02: Proceedings of the 23rd IEEE Real-Time Systems Symposium*, 2002.

[33] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008.

[34] E. L. Kaplan and P. Meier. Nonparametric estimation from incomplete observations. *Journal of the American Statistical Association*, 52 (282):457–481, 1958.

[35] E. Këllezi and M. Gilli. Extreme value theory for tail-related risk measures. Fame research paper series, International Center for Financial Asset Management and Engineering, 2000.

[36] J. Kihm, A. Settle, A. Janiszewski, and D. A. Connors. Understanding the impact of inter-thread cache interference on ILP in modern SMT processors. *The Journal of Instruction Level Parallelism*, 7, 2005.

[37] R. Kokku, T. L. Riché, A. Kunze, J. Mudigonda, J. Jason, and H. M. Vin. A case for run-time adaptation in packet processing systems. *SIGCOMM Comput. Commun. Rev.*, 34(1), 2004.

[38] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogenous multi-core architectures for multithreaded workload performance. In *ISCA '04: Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004.

[39] R. L. McGregor, C. D. Antonopoulos, and D. S. Nikolopoulos. Scheduling algorithms for effective thread pairing on hybrid multiprocessors. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, 2005.

[40] M. Norton. Optimizing Pattern Matching for Intrusion Detection, http://docs.idsresearch.org/OptimizingPatternMatchingForIDS.pdf, 2004.

[41] nProbe network monitor. http://www.ntop.org.

[42] F. Petrini, D. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, 2003.

[43] J. I. Pickands. Statistical inference using extreme value order statistics. *Annals of Statististics*, 3:119–131, 1975.

[44] P. Radojković, V. Čakarević, J. Verdú, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero. Thread to strand binding of parallel network applications in massive multi-threaded systems. In *PPoPP-2010: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Bangalore, India*, 2010.

[45] A. Settle, J. Kihm, A. Janiszewski, and D. Connors. Architectural support for enhanced SMT job scheduling. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, 2004.

[46] D. Shelepov, J. C. S. Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar. Hass: A scheduler for heterogeneous multicore systems. In *ACM SIGOPS Operating Systems Review*, 2009.

[47] T. Sherwood, G. Varghese, and B. Calder. A Pipelined Memory Architecture for High Throughput Network Processors. In *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003.

[48] E. Shmueli, G. Almasi, J. Brunheroto, J. Castanos, G. Dozsa, S. Kumar, and D. Lieber. Evaluating the effect of replacing CNK with Linux on the compute-nodes of Blue Gene/L. In *ICS '08: Proceedings of the 22nd Annual International Conference on Supercomputing*, 2008.

[49] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *ASPLOS 2000: Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[50] A. Snavely, D. M. Tullsen, and G. Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2002.

[51] Snort network intrusion prevention and detection system. http://www.snort.org/.

[52] N. Tajvidi. Design and implementation of statistical computations for Generalized Pareto Distributions. *Technical Report, Chalmers University of Technology*, 1996.

[53] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2007.

[54] TCPDUMP/LIBPCAP public repository. http://www.tcpdump.org/.

[55] S. B. Vardeman. *Statistics for Engineering Problem Solving*. PWS publishing company, 1993.

[56] V. Čakarević, P. Radojković, J. Verdú, A. Pajuelo, F. Cazorla, M. Nemirovsky, and M. Valero. Characterizing the resource-sharing levels in the UltraSPARC T2 processor. In *MICRO-42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, NY, USA*, 2009.

[57] J. Verdú. *Analysis and Architectural Support for Parallel Stateful Packet Processing, PhD Thesis*. Universitat Politècnica de Catalunya, 2008.

[58] Vermont (VERsatile MONitoring Toolkit). http://vermont.berlios.de/.

[59] S. S. Wilks. The large-sample distribution of the likelihood ratio for testing composite hypotheses. *Annals of Mathematical Statistics*, 9, 1938.

[60] S. S. Wilks. *Mathematical Statistics*. Princeton University Press, 1943.

[61] Wireshark network protocol analyzer. http://www.wireshark.org/.

[62] T. Wolf, N. Weng, and C.-H. Tai. Design considerations for network processor operating systems. In *ANCS '05: Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communication Systems*, Princeton, NJ, 2005.