Thread Assignment in Multicore/Multithreaded Processors: A Statistical Approach

Petar Radojković, Paul M. Carpenter, Miquel Moretó, Vladimir Čakarević, Javier Verdú, Alex Pajuelo, Francisco J. Cazorla, Mario Nemirovsky, and Mateo Valero, *Fellow, IEEE*

Abstract—The introduction of multicore/multithreaded processors, comprised of a large number of hardware contexts (virtual CPUs) that share resources at multiple levels, has made process scheduling, in particular assignment of running threads to available hardware contexts, an important aspect of system performance. Nevertheless, thread assignment of applications running on state-of-the art processors is an NP-complete problem. Over the years, numerous studies have proposed heuristic-based algorithms for thread assignment. Since the thread assignment problem is intractable, it is in general impossible to know the performance of the optimal assignment, so the room for improvement of a given algorithm is also unknown. It is therefore hard to decide whether to invest more effort and time to improve an algorithm that may already be close to optimal. In this paper, we present a statistical approach to the thread assignment problem. First, we present a method that predicts the performance of the optimal thread assignment, based on the observed performance of each thread assignment in a random sample. The method is based on Extreme Value Theory (EVT), a branch of statistics that analyses extreme deviations from the population mean. We also propose sample pruning, a method that significantly reduces the time required to apply the statistical method by reducing the number of candidate solutions that need to be measured. Finally, we show that, if no suitable heuristic-based algorithm is available, a sample of several thousand random thread assignments is enough to obtain, with high confidence, an assignment with performance close to optimal. The presented approach is architecture and application independent, and it can be used to address the thread assignment problem in various domains. It is especially well suited for systems in which the workload seldom changes. An example is network systems, which typically provide a constant set of services that are known in advance, with network applications performing a similar processing algorithm for each packet in the system. In this paper, we validate our methods with an industrial case study for a set of multithreaded network applications on an UltraSPARC T2 processor. This article is an extension of our previous work [44], which was published in Proceedings of 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-2012).

Index Terms—Scheduling, thread assignment, multithreading, statistical estimation, extreme value theory

1 Introduction

Multithreaded processors[1] are comprised of several cores executing threads that share resources at multiple levels [57]. For example, in a CMP processor where each core supports concurrent execution of several threads through SMT, all simultaneously running threads share global resources such as the last level of cache and the I/O. In addition to this, threads running in the same core share core resources such as the Instruction Fetch Unit, and the L1 instruction and data caches. Therefore, the way that threads are assigned to cores determines which resources they share, and this, in turn, may significantly affect the system performance. In processors with several levels of resource sharing, thread scheduling is comprised of two steps. In the first step, usually called *workload selection*, the OS selects the set of threads (workload) that will be executed in the processor in the next time slice, from a set of ready-to-run threads. In the second step, called *thread assignment*, each thread in the workload is assigned to a hardware context (virtual CPU) of the processor.

Dynamic thread 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 [24],[40] and reduce the performance provided by commercial network processors. Since maximizing the amount of computation power delivered to running parallel applications is critical to achieving high performance and scalability, many commercial systems already use Lightweight Kernels (LWKs) with static scheduling, such as CNK [50] in BlueGene HPC systems, and Netra DPS [2] which is mainly used in networking.

LWKs with static scheduling are especially well suited for network systems. Typically, these systems provide a

[1] In this paper, the term "multithreaded processor" refers 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.

- *P. Radojković, P. M. Carpenter, M. Moretó, and V. Čakarević are with the Barcelona Supercomputing Center (BSC), Barcelona, Spain. E-mail: {petar.radojkovic, paul.carpenter, miquel.moreto, vladimir.cakarevic}@bsc.es.*
- *J. Verdú and A. Pajuelo are with the Universitat Politècnica de Catalunya, Barcelona, Spain. E-mail: {jverdu, mpajuelo}@ac.upc.edu.*
- *F. Cazorla is with the Spanish National Research Council (IIIA-CSIC), Madrid, Spain, and is also with Barcelona Supercomputing Center (BSC), Barcelona, Spain. E-mail: francisco.cazorla@bsc.es.*
- *M. Nemirovsky is with the Institucio Catalana de Recerca i Estudis Avancats and is also with the Barcelona Supercomputing Center (BSC), Barcelona, Spain. E-mail: mario.nemirovsky@bsc.es.*
- *M. Valero is with the Universitat Politècnica de Catalunya and is also with Barcelona Supercomputing Center (BSC), Barcelona, Spain. E-mail: mateo@ac.upc.edu.*

Manuscript received 12 Feb. 2014; revised 17 Feb. 2015; accepted 20 Feb. 2015. Date of publication 26 Mar. 2015; date of current version 16 Dec. 2015.
Recommended for acceptance by A. Nannarelli.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2015.2417533

constant set of services which is known in advance, and network applications generally perform a similar processing algorithm for each packet in the system. Therefore, in a networking environment, the workload is typically known beforehand, and it changes infrequently at runtime. In these systems, dynamic thread scheduling is not indispensable, and the main scheduling decision is to determine a good assignment of concurrently-running threads. In general, in systems in which the workload seldom changes, finding a good thread assignment becomes the most important process scheduling problem.

In state-of-the-art multithreaded processors, finding an optimal thread assignment is an intractable problem. It is NP-complete even for simplified models of the applications and system architecture [22]. The problem is especially complicated when the number of hardware contexts (virtual CPUs) is large, or when processor resources are shared at multiple levels [47], [57]. In this case, it is hard to predict the impact of resource sharing on system performance, as shown by several studies in diverse application domains; e.g. parallel applications on massive multithreaded processors [47], multiple applications running on SMTs [16], and data centre applications on commodity CPUs [55]. As the number of possible thread assignments is vast (e.g. 10^{50}) [16], [18], [29], [45], and increases rapidly, both with the number of threads and with the number of hardware contexts, it is impractical to use exhaustive search to find the optimal thread assignment.

Over the years, numerous studies (see Section 7) have proposed heuristic-based algorithms to address the thread assignment problem. Since the thread assignment problem is intractable, it is in general impossible to know the performance of the optimal assignment, so the room for improvement of a given algorithm is also unknown. It is therefore hard to decide whether to invest effort to try to improve an algorithm that may already be close to optimal.

In this paper, we present a statistical approach to the thread assignment problem. First, we present a method that predicts the performance of the optimal thread assignment based on the observed performance of each thread assignment in a random sample. This method is based on EVT, a branch of statistics that analyses extreme deviations from the population median. We also propose and evaluate sample pruning, a method that reduces the number of random thread assignments that need to be executed on the target platform, which in turn reduces the time needed for the overall analysis. Finally, we show that the performance of the best observed thread assignment in a random sample is likely to be close to optimal, so if no suitable heuristic-based algorithm is available, a good thread assignment could be found using random sampling on its own. This paper extends our previous work [44], as discussed in Section 7.

EVT is an important branch of statistics, which has found multiple applications in civil engineering, finance and emergency planning. It provides many powerful theorems and tools, but its application in computer science and engineering is marginal. The statistical approach described in this paper can be applied to other intractable problems in many diverse fields of computer science and engineering. In fact, we have already applied it successfully to compilation of multithreaded streaming applications [43]. The statistical analysis infers the population maximum (or minimum) based on a random sample, in a way that is independent of the problem being addressed. As such, it does not require a profound understanding of the target system, so it can be employed without a significant investment in effort or time. The method is particularly useful in the evaluation of any new proposed heuristics-based algorithm.

The rest of this paper is organized as follows. Section 2 describes the application domain and motivates the need for static thread assignment. Section 3 presents the experimental environment used in our study. Section 4 presents the statistical method that estimates the optimal system performance, with an emphasis on understanding and intuition. Section 5 proposes sample pruning, which reduces the time needed to apply the statistical method. Section 6 presents the random sampling method, which can find a good solution when no suitable heuristic is available. Section 7 describes the related work, and Section 8 presents the conclusions. Finally, the Appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TC.2015.2417533 available online, includes detailed description of the benchmarks and the statistical method.

2 Background

In this paper, we focus on the problem of thread assignment for network applications running on multithreaded processors. Network applications are increasing in importance, with the increasing number and complexity of Internet services, and the tremendous growth in Internet traffic. Providing high-performance network services is critical to sustaining online services and avoiding packet drop, since network processors are saturated by the high network bandwidth, and it is constantly increasing.

In order to provide high throughput and low latency, network applications impose specific requirements on the hardware and operating system. Network applications exhibit significant parallelism at multiple levels [58], so they are well suited to run on massively multithreaded processors. These processors are able to process numerous packets concurrently, through the simultaneous execution of a large number of hardware contexts.

Network-oriented systems require run-time environments that provide high performance, high-speed packet processing and predictable execution time. To that end, these systems use low-overhead low-noise run-time environments, which reduce the performance impact of system overhead incurred by management threads [2]. Such runtime environments reduce system overhead by providing only the essential system services. One feature omitted by several widely used lightweight runtime environments is dynamic thread scheduling.

In networking environments, threads are typically mapped to hardware contexts statically, i.e. before runtime. This is because network applications typically perform a fixed processing algorithm, applied to each packet that arrives, which is known beforehand and seldom changes over time.

Optimal thread assignment requires a deep knowledge of the resource requirements of the various threads and an understanding of how the threads interact at each shared processor resource. It is difficult to determine the optimal thread assignment without running many experiments, the number of which rapidly increases with the number of hardware contexts, amount of resource sharing, and the number of concurrently running threads. The problem is even harder when the applications themselves are multithreaded, since to determine the bottlenecks, the designer must be aware of how the threads communicate.

Thread assignment is typically done in one of three ways:

1) *Manual assignment.* A skilled designer determines a good thread assignment based on a detailed analysis of the target architecture and offline profiling [57]. The 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 threads. Manual thread assignment is expensive and time consuming, and any change in the application or hardware platform requires the whole analysis to be repeated.

2) *Performance predictors.* Numerous studies propose methods to predict the performance of different thread assignments for a given workload, based on architecture-dependent heuristics (see Section 7). Since the number of possible thread assignments is huge, it is infeasible to predict the performance of all assignments. The predictors are therefore used to estimate performance for a sample of assignments and determine the best assignment in the sample. In addition, the predictors introduce an error when estimating performance, so the assignment with the best predicted performance may not be the actual best one. Another drawback of most currently-available performance predictors is that they do not support multithreaded applications.

In both, manual thread assignment and performance predictors, a change in architecture or workload may require significant extra time and effort to recalculate the thread assignment. When thread assignment is done manually, the designer must repeat the analysis. When using performance predictors, the prediction algorithm may require significant changes.

3) *Load balancing and cache affinity.* State-of-the-art fully-fledged OSs, such as Linux and Solaris, use a load balancing mechanism [3], [56] to equally distribute the load of the running threads among all available scheduling domains (e.g. cores of the CMP architecture). In addition, cache and TLB affinity algorithms [3], [56] keep each thread assigned to a single logical CPU in order to avoid cache and TLB misses that would be caused by thread migration. Although these techniques improve performance, they are insufficient to fully exploit the capabilities of current multithreaded processors with several levels of resource sharing. Load balancing does not take into account the differing amounts of processor resources used by the various co-running threads, and therefore it may ignore the best schedules, considering them to be "unbalanced" [16]. In our previous studies [44], [45], [46], we evaluate load-balancing thread assignment for networking applications running on the UltraSPARC T2 processor, measuring a significant performance loss compared with the optimal assignment, of up to 48 percent.

Importance of knowing the optimal system performance. Numerous studies present algorithms for thread assignment on multithreaded processors (see Section 7). A new proposal would, in an ideal world, be compared with the optimal solution, but, in general, the optimal cannot be found without running all valid thread assignments. Several authors [4], [18], [45], therefore, verify their proposals with respect to either a naive thread assignment, in which threads are randomly assigned to the virtual CPUs of the processor, or Linux-like assignments, in which the number of threads per core or per scheduling domain is balanced. It is our position that the evaluation of these proposals would be significantly improved if they were compared to an estimate of the optimum. This argument is described in more detail in the ASPLOS publication [44].

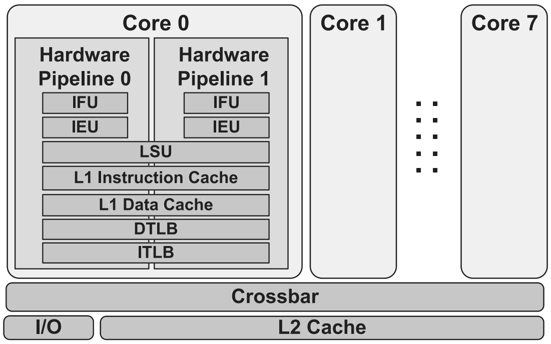


Fig. 1. Schematic view of the three resource sharing levels of the UltraSPARC T2 processor.

3 Experimental Environment and Methodology

We evaluated the statistical analysis using multithreaded network applications running in a real industrial environment. We used two SPARC Enterprise T5220 servers, each of which contained one UltraSPARC T2 processor. One T5220 machine generated the network traffic using the Oracle Network Traffic Generator (NTGen) [2]. NTGen is a software tool that produces IPv4 TCP/UDP packets, with a set of configuration parameters controlling various packet header fields. This machine was connected via a 10Gb link to the second T5220 machine, on which we executed the selected thread assignments. In all experiments presented in the study, we verified that NTGen generated sufficient traffic to saturate the network processing machine. The performance bottleneck was therefore the packet processing speed, which is determined by the performance of the selected thread assignment.

3.1 UltraSPARC T2 Processor

The UltraSPARC T2 is a multithreaded processor [1] with eight cores connected through a crossbar to a shared L2 cache (see Fig. 1). Each core supports eight hardware contexts, four on each execution pipeline, giving a total of 64 hardware contexts for the entire processor. Threads may, therefore, share (and compete for) hardware resources at three different levels: *IntraPipe*, *IntraCore*, and *InterCore*. Resources at the *IntraPipe* level, such as the Instruction Fetch Unit (IFU) and the Integer Execution Units (IEU), are shared among threads running in the same hardware pipeline. The *IntraCore* resources, such as the Load Store Unit (LSU), L1 instruction cache, L1 data cache, data and instruction TLBs, as well as Floating Point and Graphic Unit (FPU), and Cryptographic Processing Unit, are shared among threads running on the same core. Finally, the *InterCore* resources, including the L2 cache, on-chip interconnection network (crossbar), memory controllers, and the interface to off-chip resources, are shared among all threads running on the processor [57].

3.2 Netra DPS

Networking systems use lightweight runtime environments to reduce the overhead that would be introduced by a fully-fledged OS [42]. One of these environments is Oracle's Netra DPS [2]. In order to reduce overhead and noise, Netra DPS omits certain common OS features, including virtual memory, interrupt handling, daemons, context switching, and the run-time process scheduler. A thread always runs to completion on its assigned hardware context, without preemption. The assignment of running threads to processor hardware contexts (virtual CPUs) must therefore be performed statically at compile time. It is the responsibility of the programmer or toolchain to determine which hardware context will execute each particular thread.

3.3 Benchmarks

This section briefly describes the benchmarks that we used in this study. Detailed presentation of the benchmarks is included in Appendix A, available in the online supplemental material. Note that, since Netra DPS omits standard OS features including dynamic memory allocation and file management, we had to adapt some of the benchmarks to execute in this environment.

The benchmarks used are described next:

1) IP Forwarding (IPFwd) application decides where to forward a packet for 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, IPFwd may exhibit significantly different memory behavior. In order to cover different cases of IPFwd memory behavior, we created two variants of the IPFwd application, both based on the IPFwd application included in the Netra DPS distribution [2]: (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).

2) Packet analyzer is a program that can intercept and log traffic passing over a network or part of a network [15]. The packet analyzer 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.

3) Aho-Corasick is a string matching algorithm. String matching is the basic technique to analyze network traffic at the application layer. In the experiments presented in this paper, we used the Aho-Corasick algorithm to search the packet payloads for the keywords in the Snort Denial-of-Service set of intrusion detection rules (version 2.9).

4) Stateful packet processing is an important component of state-of-the-art network monitoring tools and intrusion prevention and detection systems. Unlike stateless applications, which process packets independently (examples include the IPFwd, Packet analyzer, and Aho-Corasick benchmarks above), stateful packet processing keeps information from the processing of previous packets.

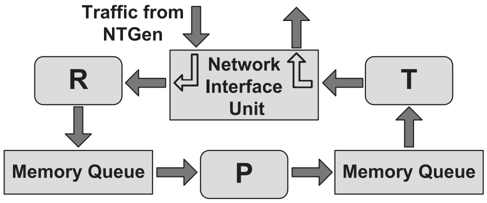


Fig. 2. The schematic view of the benchmarks.

Benchmark implementation. As shown in Fig. 2, each benchmark is divided into three threads: Receiving (R), Processing (P), and Transmitting (T). R, P, and T threads communicate through memory queues and process network packets in a pipelined fashion. The workload consists of several benchmark instances running concurrently, each of which has the R, P, and T threads. This is a common way to implement network applications [2], [62].

Summary. The presented benchmarks represent a good testbed for the analysis of thread assignment techniques because:

1) Each benchmark instance has three different threads, so even when the workload consists of several instances of the same benchmark, the system must deal with heterogeneous threads.
2) The benchmarks stress the hardware resources of the UltraSPARC T2 processor at all three sharing levels [57].
3) Each benchmark instance has threads that communicate through shared memory queues. The benchmark performance also depends on the distribution of interconnected threads among processor cores (L1 cache domains).
4) Thread assignment has a significant impact on performance. We detect performance variation of up to 49 percent between different thread assignments of the same workload.

3.4 EVT Requirements and Sampling Methods

EVT has two main prerequisites, which should be validated before applying the theory to a particular real-life problem [25]. The first prerequisite is that the sample under study is comprised of independent and identically distributed (i.i.d.) observations. The second prerequisite is that the probability distribution of the statistics under study should be continuous. These requirements and the statistical tests used for their evaluation are discussed in detail in Appendix B.2, available in the online supplemental material.

3.5 Methodology

In all experiments, we simultaneously executed eight benchmark instances, giving 24 threads. We could not execute more than eight benchmark instances because of a limitation in the experimental environment: the on-chip NIU of the UltraSPARC T2 processor can split the incoming network traffic into up to eight DMA channels, and, under Netra DPS, each DMA channel can be bound to at most one receiving thread.

In order to ensure stable results, we measured the execution time to process three million network packets per benchmark instance. This means that each application

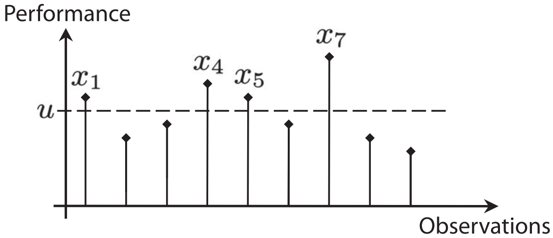


Fig. 3. Extreme values over the threshold u.

thread had a loop that executed three million times. The execution time of each experiment was around 1.5 seconds, with the precise duration depending on the benchmark and on the performance of the thread assignment under test.

4 Estimation of the Optimal Performance—POT Method

The best way to evaluate any thread assignment approach is to compare the performance of the thread assignment provided by the approach with the performance of the optimal assignment, i.e. with the optimal system performance. The difference in performance gives the maximum potential for improvement of the proposed scheduling approach. Since thread assignment is NP-complete and the number of possible thread assignments is vast, the optimal system performance cannot be determined [29]. 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 thread assignments.

4.1 Peak over Threshold (POT) Method

We estimate the performance of the optimal thread assignment using EVT. EVT is a branch of statistics that studies extreme deviations from the population median [7], [10]. One of the EVT approaches is the POT method. The POT method takes into account the distribution of the observations that exceed a given (high) threshold. For example, in Fig. 3, 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 be explained using the cumulative distribution function (CDF). For example, assume that F is the CDF of a random variable X, which is defined as $F(x) = \mathrm{P}(X \leq x)$. 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:

$$F_u(y) = \mathrm{P}(X - u \leq y | X > u),\ 0 \leq y \leq x_F - u,$$

where X is the observed random variable, u is the 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. Fig. 4 shows the CDF of a random variable X (upper chart) and the corresponding conditional excess distribution function $F_u(y)$ (lower chart).

The POT method is based on the *Pickands-Balkema-de Haan* theorem [6], [41]:

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

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

for $y \in [0, (x_F - u)]$ if $\xi \geq 0$ and $y \in [0, -\frac{\sigma}{\xi}]$ if $\xi < 0$.

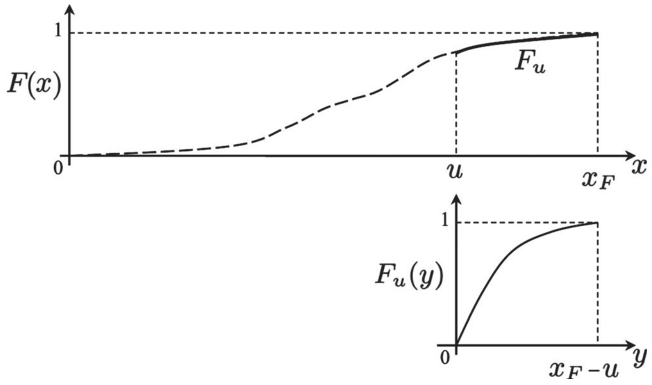


Fig. 4. Cumulative distribution function $F(x)$ and corresponding conditional excess distribution function $F_u(y)$.

4.2 Application of POT to Thread Assignment

Theorem 1 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 Generalized Pareto Distribution (GPD). GPD is defined with two parameters: shape parameter ξ and scaling parameter σ. An important characteristic of GPD used in this study is that for $\xi < 0$, the upper bound of the observed value (in our study, the performance of the optimal thread assignment) can be computed as $u - \frac{\sigma}{\xi}$, where u is the selected threshold and σ and ξ are the GPD parameters [23], [30].

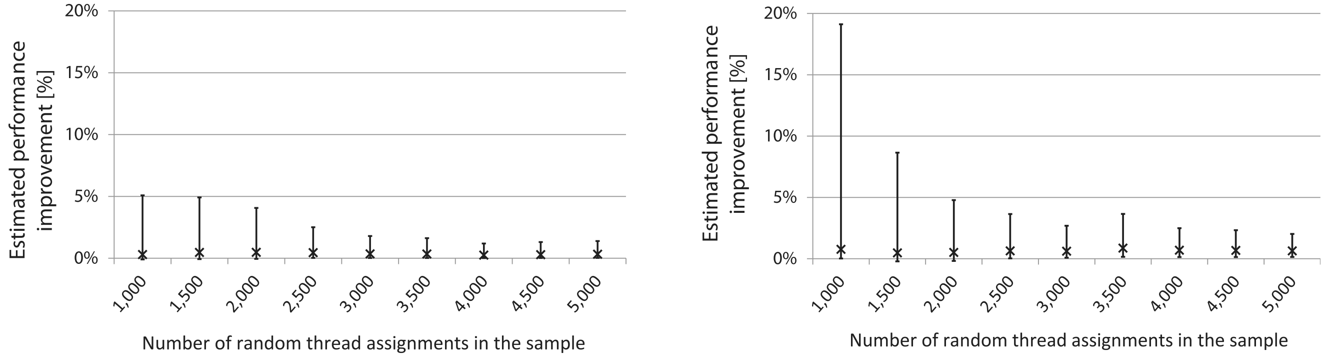
We use the POT method to estimate the optimal system performance, i.e. the performance of the optimal thread assignment, for a given workload based on the measured performance of the sample of random thread assignments. Application of POT method to the thread assignment problem is explained in detail in Appendix B, available in the online supplemental material. In this section, we present a brief intuitive overview of the main steps of the analysis.

The application of the POT method to the thread assignment problem involves the following four steps:

Step 1*: Generate the sample of random thread assignments, execute the assignments on the target machine, and measure the performance of each assignment.* The method used to generate random thread assignments and experimental methodology are described in detail in Appendix B.2, available in the online supplemental material, and Section 3.5, respectively.

Step 2*: Select the threshold u.* Selection of the threshold u is an important step in POT analysis. In this study, the threshold u is selected using graphical methods: sample mean excess plot [23], [30] and quantile plot [7], [30].

Step 3*: Fit the GPD function to the observations that exceed the threshold and estimate parameters ξ and σ.* Once the threshold u is selected, the observations over the threshold can be



(a) Aho-Corasick benchmark

(b) IPFwd-L1 benchmark

Fig. 5. Impact of the sample size on the estimation of the optimal performance.

fitted to GPD, and the parameters of the distribution can be estimated. Different methods can be used to estimate the parameters of GPD from a sample of observations [9], [26], [27], [53]. In our study, GPD parameters were estimated using the likelihood function, a statistical method that estimates distribution parameters based on a set of observations [5].

Step 4*: Estimate the optimal system performance, i.e. the upper performance bound of all thread assignments.* The point estimate of the Upper Performance Bound (UPB) is computed as $\widehat{\mathrm{UPB}} = u - \hat{\sigma}/\hat{\xi}$, where $\hat{\sigma}$ and $\hat{\xi}$ are estimated values of the GPD parameters. 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. In addition to the UPB point estimate, in order to indicate the confidence of the estimate, we compute the confidence intervals of the estimated UPB. UPB confidence interval is computed using likelihood ratio test [5] and Wilks's theorem [13], [60], [61].

4.3 POT Evaluation

This section evaluates the POT statistical method by applying it to the thread assignment problem for multithreaded network applications on the UltraSPARC T2 processor. Also, we determine how many random thread assignments are required to obtain a precise estimate of the optimal performance.

4.3.1 Applicability of the POT Method to the Thread Assignment Problem

There are several reasons why the POT method may fail to produce an estimate. First, the exceedances may not fit to a GPD. This can be easily detected in Step 2 and Step 3 of the POT analysis (see Appendix B.3, available in the online supplemental material). Second, the upper bound of the estimated optimal performance may diverge to infinity ($\xi \geq 0$). Finally, the estimation of the optimal system performance (Step 4 of the POT method) is based on an iterative method that may not converge to a solution (for more details, see Appendix B.3, available in the online supplemental material).

Therefore, the first step of our evaluation is to check whether or not the POT method is able to produce an estimate of the optimal performance. For each benchmark, we generated a uniformly distributed sample of 5,000 random thread assignments and executed them on the UltraSPARC T2 processor. We then attempted to apply the POT method to the 5,000 observed performance measurements. The method *successfully produced an estimate of the optimal performance, for all benchmarks under study*.

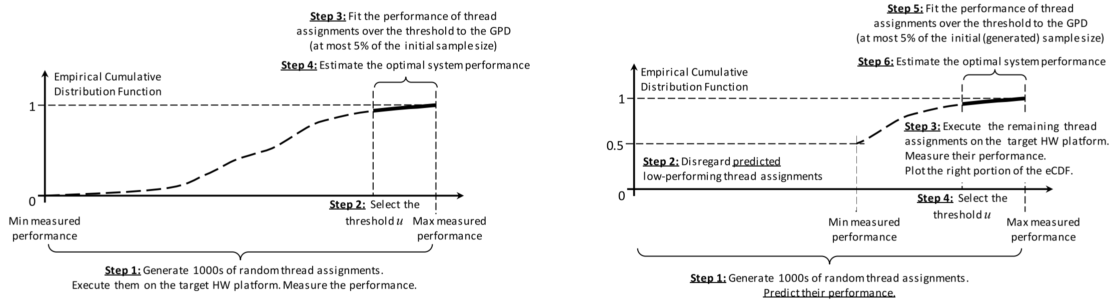
4.3.2 Precision of the Estimation

In order to understand how the precision of the estimate depends on the sample size, we varied the number of thread assignments in the random sample between 1,000 and 5,000, and estimated the optimal performance for different sample sizes. Samples that comprise from 1,000 to 4,500 observations are selected as a random subset from the complete sample of 5,000 thread assignments.

Intuitively, we would expect that, as the sample size is increased, the precision of the performance estimate would also increase. In general, larger samples contain more thread assignments with performance above the threshold, so that more data values are fitted to the GPD, and consequently the estimated GPD parameters and the optimal performance are all more precise. This is what we observe. *For all the benchmarks used in the study, we detected that the width of the confidence bounds reduces as the number of thread assignments in the sample increases.* We also noticed that, as the sample size increases, the point estimation of the optimal performance remains roughly the same, and the confidence bounds converge to this value.

Fig. 5 shows more detail for two illustrative benchmarks: *Aho-Corasick* and *IPFwd-L1*. In each chart, the x-axis lists the sample size, while the y-axis shows the estimated performance improvement—the relative difference between the estimated optimal performance and the actual performance of the best thread assignment in the random sample. The cross markers correspond to the point estimation of the optimal performance, and the error bars show the confidence bounds for the 0.95 confidence level.

For the *Aho-Corasick* benchmark, with 1,000 random thread assignments, the width of the 0.95 confidence interval is around 5 percent, which is still sufficient to estimate the optimal performance with high precision. On the other hand, for the *IPFwd-L1* benchmark, an estimation based on 1,000 random thread assignments has wide confidence bounds of almost 20 percent, as can be seen in Fig. 5b. For this benchmark, precise estimation of the optimal performance requires at least 2,000 thread assignments. Results for the *IPFwd-Mem*, *Packet Analyzer*, and *Stateful* benchmarks follow a similar trend as that of the *IPFwd-L1* benchmark (Fig. 5b).


Step 3: Fit the performance of thread assignments over the threshold to the GPD (at most 5% of the initial sample size)
Step 4: Estimate the optimal system performance
Empirical Cumulative Distribution Function
1
0
Step 2: Select the threshold u
Min measured performance
Max measured performance
Step 1: Generate 1000s of random thread assignments. Execute them on the target HW platform. Measure the performance.
Step 5: Fit the performance of thread assignments over the threshold to the GPD (at most 5% of the initial (generated) sample size)
Step 6: Estimate the optimal system performance
Empirical Cumulative Distribution Function
1
0.5
0
Step 3: Execute the remaining thread assignments on the target HW platform. Measure their performance. Plot the right portion of the eCDF.
Step 2: Disregard predicted low-performing thread assignments
Step 4: Select the threshold u
Min measured performance
Max measured performance
Step 1: Generate 1000s of random thread assignments. Predict their performance.

(a) Baseline POT algorithm

(b) Sample-pruning SP-POT algorithm

Fig. 6. Application of the statistical methods to the thread assignment problem.

We therefore conclude that the appropriate sample size for precise estimation of the optimal performance depends on the benchmark under study. If the user requires a particular precision, we propose the following iterative method. The user first generates a small sample, of approximately 1,000 thread assignments, and produces an estimate of the optimal performance using the POT method. If the precision of the estimate does not satisfy the user's requirements, then the sample size should be increased and the analysis repeated, until it does.

5 Sample Pruning—SP-POT Method

Our analysis of the time needed for the POT method shows that most of that time was spent on executing random thread assignments on the UltraSPARC T2 processor. This section proposes *sample pruning*, a method that reduces the time required for the statistical analysis, by executing a smaller number of thread assignments on the target machine. We also present the Sample Pruning POT (SP-POT) method, which integrates sample pruning into the POT method. We then evaluate SP-POT performance, finding an eightfold reduction in analysis time for a negligible difference in the estimate.

5.1 SP-POT Motivation

The POT method requires performance measurements for thousands of random thread assignments. Obtaining this data requires that each thread assignment is executed on the hardware platform under study, which involves, in total, a significant amount of execution time. For example, in our study, executing 5,000 random thread assignments on the UltraSPARC T2 processor took, depending on the benchmark, between 1.5 and 2.5 hours. On the other hand, the remaining steps of the analysis can be completed in a few minutes: generating 5,000 uniformly distributed random thread assignments takes less than a minute, and the POT statistical analysis is done within a similar time. This means that, in our study, more than 95 percent of the time needed for the overall analysis was expended on executing the random thread assignments on the hardware platform.

In order to examine the opportunity to reduce the experimentation time, we analyze the POT algorithm presented in Section 4. Fig. 6a shows an overview of the POT algorithm from the perspective of the Empirical Cumulative Distribution Function (ECDF) of the performance measurements on the hardware platform. The x-axis gives the performance of the thread assignments, from the minimal to the maximal observed in the sample. The y-axis is the fraction of the thread assignments in the sample with performance less than or equal to the value on the x-axis.

In the POT algorithm, Steps 3 and 4, which fit the GPD function to the observations then estimate the optimal system performance, only depend on the thread assignments that exceed threshold u, as does calculating the confidence interval using the method described in the appendix, available in the online supplemental material. Since the GPD is fitted to the tail of the distribution, this threshold is typically chosen at the 95th percentile or above [23],[30], meaning that only about 5 percent of the thread assignments are used in Steps 3 and 4 (see Fig. 6a). The remaining thread assignments, which comprise at least 95 percent of the execution time, are disregarded from the rest of the analysis.

5.2 Sample Pruning and SP-POT Algorithm

We propose *sample pruning*, a method that reduces the number of random thread assignments that need to be executed on the target platform, which in turn reduces the time to apply the overall analysis. Both the sample pruning technique and the enhanced version of the POT algorithm, which is known as SP-POT, are illustrated in Fig. 6b. This figure plots the ECDF of the performance measurements, in a similar way to Fig. 6a. Application of the POT statistical method *with sample pruning* to the thread assignment problem is comprised of the following six steps:

Step 1*: Generate the sample of random thread assignments, and* ***predict the performance*** *of each of them.* The performance of a given thread assignment can be predicted using a heuristic based on analysis of the application hardware requirements and properties of the target platform. Previous studies that address the process scheduling problem for multithreaded processors [20], [45], [46], [51], [52] present several systematic methods to predict the performance of workloads executed on multithreaded processors. Any such method may be used to predict the performance of the randomly-generated thread assignments.

Step 2*: Disregard predicted low-performing thread assignments.* Thread assignments predicted to have low or medium performance are unlikely to be in the upper tail of the ECDF, i.e. they are unlikely to be in the 5 percent of the best-performing assignments, and therefore unlikely to be fitted to the GPD. We therefore discard them from further analysis. In this study, we analyze three levels of sample pruning, by eliminating the thread assignments with predicted performance in the bottom 50, 75, and 90 percent, respectively.

***Step 3**: Execute the remaining thread assignments on the target hardware platform, and measure their performance.* These values are used to create the right-hand portion of the ECDF, as illustrated in Fig. 6b. This figure shows the case where 50 percent of the thread assignments are to be executed on the real hardware. As before, the x-axis of the ECDF ranges from the minimal to maximal performance, but as measured in the *pruned* sample (the diagram has the same scale as before only for simplicity in presentation). Since the ECDF is plotted for the 50 percent of thread assignments with medium to high predicted performance, the y-axis ranges from 0.5 to 1.

***Step 4**: Select the threshold u.*

***Step 5**: As before, fit the GPD function to the performance observations that exceed the threshold, and estimate parameters ξ and σ.*

***Step 6**: Produce the point estimate and confidence bounds of the optimal system performance, as before.*

Steps 4, 5, and 6 of the SP-POT algorithm, which includes sample pruning, are exactly the same as Steps 2, 3, and 4 of the baseline POT algorithm from Section 4 (compare also Fig. 6a and 6b).

5.3 SP-POT Evaluation

This section evaluates the SP-POT method by applying it to all the benchmarks under study, in a similar way to the analysis of the POT method in Section 4.3. The benchmarks were executed in the environment that comprises UltraSPARC T2 processor and Netra DPS low-overhead runtime environment, as described in Section 3. In all the experiments, we simultaneously executed 24 software threads, the maximum number that could be executed in the current experimental environment (see Section 3.5).

In Step 1 of SP-POT algorithm, we predicted the performance of the thread assignments using the *BlackBox scheduler* [46]. The BlackBox scheduler is a model that predicts the performance of different thread assignments of applications running on multithreaded processors. There are three principal reasons that motivated us to choose this performance predictor. First, the BlackBox scheduler is one of the few methods designed to predict the performance of *multithreaded* applications. Second, the model can be easily adapted to different applications, since it requires no data about the hardware requirements of the applications under study, nor does it require any modifications to the application source code. Third, the model requires minimal information about the target processor architecture and it can be easily applied to different hardware platforms.

Our analysis has three main goals. First, we evaluate whether or not the SP-POT algorithm with sample pruning could produce a performance estimate of the optimal thread assignment. Second, we quantify the error that is introduced by sample pruning. Finally, we analyze the speedup in analysis time, compared with the baseline POT algorithm.

For each of the benchmarks under study, we applied the SP-POT statistical algorithm to the same sample of 5,000 randomly-generated thread assignments used in Section 4.3. We applied three levels of sample pruning, at the 50, 75, and 90 percent levels, meaning that the indicated proportion of thread assignments were discarded, leaving 50, 25, or 10 percent of them to be executed on the UltraSPARC T2 processor. *The SP-POT algorithm successfully estimated the performance of the optimal thread assignment in all the experiments*.

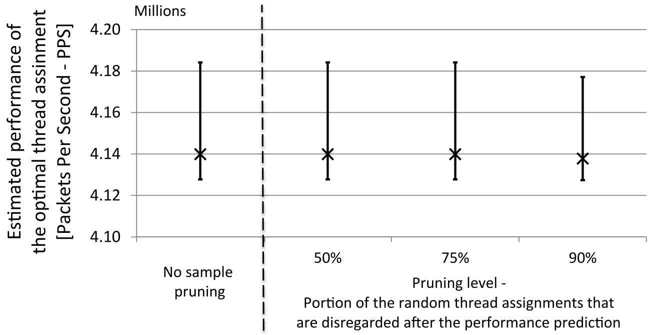


Fig. 7. Sample pruning evaluation (Aho-Corasick).

Fig. 7 quantifies the estimation error introduced by sample pruning, in the case of the *Aho-Corasick* benchmark. This benchmark is representative of our results, since we observed similar behaviour for all benchmarks under study. The figure plots the point estimate and confidence bounds for the baseline POT algorithm, with no sample pruning, and for the SP-POT algorithm at multiple pruning levels. The algorithm and pruning levels are indicated on the x-axis. When 50 or 75 percent of the thread assignments with predicted low/mid performance were discarded from the analysis, the optimal performance estimate matched the estimation of the baseline POT algorithm. At the 90 percent pruning level, i.e. in the experiment in which only 10 percent of the generated random thread assignment were executed on the hardware platform, the error introduced by sample pruning is still negligible. The error of the point estimation is 2,100 packets per second (PPS) or 0.05 percent. The error of the estimated confidence bound width is 6,700 PPS, which coresponds to 0.16 percent of the point estimate.

Finally, we compare the time needed for the overall statistical analysis with and without sample pruning. Generating 5,000 random thread assignments required less than a minute. Executing the thread assignments, for the *Aho-Corasick* benchmark on the UltraSPARC T2 processor, required around two hours. Finally, the POT analysis is done in approximately two minutes. The overall analysis for the *Aho-Corasick* benchmark without any sample pruning therefore required approximately 2 hours and 3 minutes.

Sample pruning did not affect the time needed to generate the random thread assignments, nor did it affect the time for the POT statistical analysis. The SP-POT algorithm requires an additional step: prediction of the performance of each randomly-generated thread assignment. In our experiments, the BlackBox scheduler predicted the performance of all 5,000 thread assignments in about two seconds. This time was roughly the same for all benchmarks under study. On the other hand, the time required to run the random thread assignments was roughly proportional to the number of thread assignments executed, so it reduced significantly. For example, for the *Aho-Corasick* benchmark at the 90 percent pruning level, all experiments on the UltraSPARC T2 processor were executed in less than 12 minutes. The overall analysis time using the SP-POT method at the 90 percent sample pruning level was therefore around 15 minutes, eight times faster than the baseline POT method.

The analysis time for a given benchmark depends on how long it takes to execute on the target hardware platform. Nevertheless, all benchmarks under study gave results comparable with those presented for *Aho-Corasick*.

To summarize, the presented case study showed that sample pruning is a promising way to significantly reduce the time required to apply the POT method to the thread assignment problem. We showed that even aggressive pruning at the 90 percent pruning level introduced a negligible estimation error. In this case, applying the sample pruning technique to the *Aho-Corasick* benchmark reduced the overall analysis time from more than 2 hours to 15 minutes.

5.4 Excessive Pruning

Since at most 5 percent of the best-performing thread assignments are fitted to the GPD, one may consider a more aggressive level of pruning, by executing just 5 percent of the thread assignments on the target platform. This approach, however, did not provide a good optimal performance estimate.

The main reason for this was the prediction error of the BlackBox scheduler. The BlackBox scheduler is a simplified model, so, like all performance predictors, it is subject to error. Therefore, some of 5 percent of the *predicted* best-performing thread assignments were not within the *actual* best-performing ones. The performance prediction error of the BlackBox scheduler significantly altered the shape of the datasets used by the POT statistical method. The SP-POT method at the 5 percent level failed to produce a performance estimate, for all benchmarks under study.

In general, it is important to note that the accuracy of SP-POT depends on the accuracy of the performance predictor. Evaluation of multiple performance predictors and their suitability for sample pruning is an interesting avenue of future work.

6 Random Sampling Approach to Thread Assignment

In this section, we analyze whether a good thread assignment can be found by taking the best thread assignment from the random sample. This approach is especially useful if a heuristic-based algorithm is not available for the user's problem. First, we compute the probability that a random sample of n thread assignments captures at least one of the $p\%$ of the best-performing assignments, where p is between 0 and 100. We then evaluate the random sampling approach by comparing its performance with the estimate of the optimal thread assignment produced by the statistical techniques.

6.1 Probability that Random Sampling Detects a Good Thread Assignment

The probability that a sample of random assignments selected from a vast population contains the assignment with the *optimal* 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 thread assignment from the $p\%$ of best-performing assignments. Event A' is the opposite of event A, representing the probability that the random sample contains zero thread assignments from $p\%$ of the best-performing assignments. If the number of possible thread 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 thread 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 [14]. In addition to this, we assume that the selected thread 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, since $P(A) + P(A') = 1$. Therefore, the probability of the event A can be computed as:

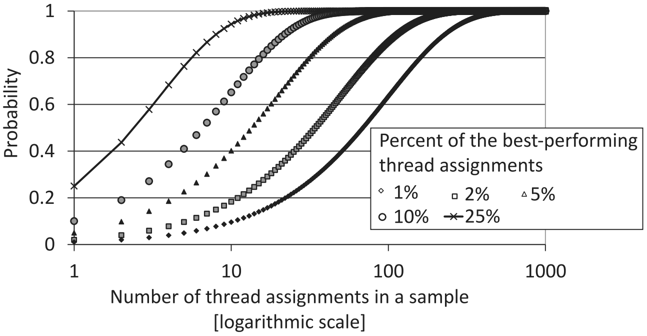


Fig. 8. Probability that a sample contains a thread assignment from $p\%$ of the best-performing assignments.

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

We observe that the probability that a sample of random thread 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 thread assignments). However, we have to be aware that this is valid only for large populations with uniform sampling, which is satisfied in the case of thread scheduling problems in state-of-the-art multithreaded processors [44].

Fig. 8 plots the probability $P(A)$ for the samples of different size and for different percentages of the best-performing thread 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 thread assignments. The figure shows data for p = 1, 2, 5, 10, 25. We derive three conclusions from Fig. 8. First, that the probability asymptotically approaches 1 as the number of thread assignments in the sample increases. Second, as the fraction of the best-performing assignments decreases (from 25 to 1 percent in the figure) the probability approaches 1 slower (more thread assignments are required to reach a high probability). Finally, we observe that small samples of below 10 elements are unlikely to capture any thread assignment from 1, 2, and 5 percent of the best-performing ones. However, a

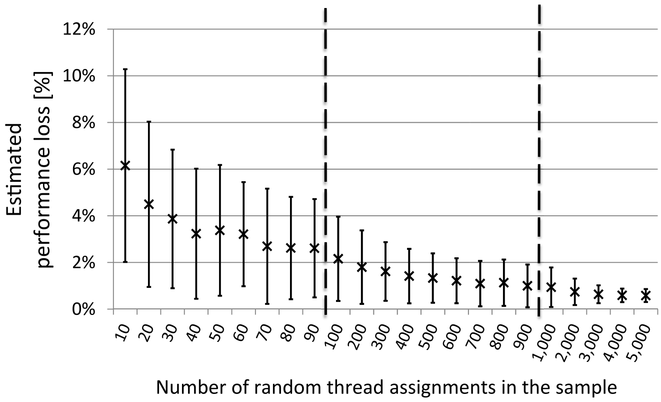


Fig. 9. Impact of sample size on the performance of the random sampling approach (Aho-Corasick).

sample of several hundred random observations is sufficient to capture at least one of 1 or 2 percent of the best-performing thread assignments with a very high probability. This means that, if we assume that 1 or 2 percent of the best-performing assignments have a good performance, simply running several hundred or several thousand randomly selected thread assignments is sufficient to capture at least one assignment with a good performance.

6.2 Random Sampling Approach Evaluation

This section evaluates the random sampling approach. We follow the experimental methodology described in Section 3, executing the same benchmarks on the UltraSPARC T2 with the Netra DPS low-overhead runtime environment. As before, for all benchmarks, we executed 24 software threads, the maximum possible with the current experimental environment (see Section 3.5).

In order to evaluate the random sampling approach, we vary the sample size, and compare the maximum observed performance with the point estimate from the POT method (with the largest sample size). Fig. 9 shows the results for the *Aho-Corasick* benchmark. The sample size varies along the x-axis of the figure. Dashed vertical lines separate the results for tens (from 10 to 90), hundreds (from 100 to 900), and thousands (from 1,000 to 5,000) of random thread assignments. The y-axis shows the relative difference between the maximal performance captured in the random sample and the POT point estimate based on the sample of 5,000 thread assignments.

In order to present meaningful results, each experiment was repeated 100 times, each using a different uniformly distributed sample. Repeating the experiments 100 times was found to be sufficient for statistical analysis of average behavior. The figure shows the mean performance loss, compared with the POT point estimate (cross marker), together with the standard deviation (error bars).

For tens of random thread assignments in the sample, we detect a moderate performance loss. We also detect a relatively high deviation, which indicates that the performance of the best-captured thread assignment is significantly different for different samples of the same size. As the sample size increases, the standard deviation decreases, which means that different samples of the same size provide similar performance. Also the best captured performance slowly converges to the estimated optimal one. Finally, several hundred or several thousand random thread assignments capture performance that is very close to the estimated optimal one, with low standard deviation of between 1 and 2 percent. We detect the same trend for all the benchmarks under study. Therefore, we conclude that uniformly distributed random sampling can be used to find a good thread assignment. However, we recommend this method only when the random sample contains at least several hundred or several thousand assignments.

7 Related Work

Workload Selection. Several approaches addressing the workload selection problem propose models that predict the impact on system performance of interference among co-running tasks. Snavely et al. [51], [52] 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 [20] propose the probabilistic job symbiosis model, which enhances the SOS scheduler. Based on the cycle accounting architecture [19], [36], [37], the model estimates the single-threaded progress for each job in a multithreaded workload. Other approaches [12], [21], [31], [48] propose techniques to construct workloads of tasks that exhibit good symbiosis in shared caches solving problems of cache contention.

Zhan et al. [63], [64] propose cache management techniques that consider spacial partitioning and replacement policy. First, these techniques maximize the cache utilization by spatially partitioning the cache resources between simultaneously-running threads. The cache replacement policy for each running thread is adjusted to optimally utilize its cache resources.

Doucette and Fedorova [17] classify applications on the basis of their usage of shared processor resources. The authors co-schedule applications with *base vectors* (micro-benchmarks that specifically stress a single CPU resource), and measure the slowdown that the application and base vector experience. This data is used to predict the performance of any set of applications running concurrently on the processor.

Kwok and Ahmad [34], [35] present a survey and an extensive performance study of different scheduling algorithms for multithreaded applications running on clusters of interconnected single-threaded processors. Since each thread is executed on a single-threaded processor, co-running threads do not collide in processor resources. Therefore, the presented scheduling algorithms do not analyze inter-thread interference in shared processor resources, which is the focus of our study.

Thread 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 thread assignment [4], [18], [45], [55], [59]. El-Moursy el al. [18] focus on multithreaded processors and propose an algorithm that uses hardware performance counters to profile thread behavior and assign compatible tasks on the same SMT core.

Tang et al. [55] study the impact of sharing memory resources (L2 and L3 cache memory, front side buses, and a memory controller hub) on Google datacenter applications running on multisocket/multicore servers. The authors propose two thread assignment approaches, and demonstrate that advanced thread assignment methods may significantly improve the performance of state-of-the-art datacenters.

Our previous studies [45], [46] address the problem of thread assignment of multithreaded network applications running on processors with several levels of resource sharing. We focus on thread assignment for lightweight kernels with static scheduling, systems that are especially well suited for network applications.

Jahn et al. [28] analyze the performance of software-pipelined applications running on many-core systems. The study addresses a challenging scenario in which the applications show unpredictable and significant variances in the demand of hardware resources.

Bulpin et al. [8] modify an existing OS scheduler to calculate dynamic priority of active threads using data from the hardware performance counters. Parekh et al. [39] show the importance and the benefit of scheduling that takes into account process characteristics. The authors propose an algorithm that reads the values of thread-specific hardware performance counters to estimate characteristics of running threads. De Vuyst et al. [16] explore thread scheduling for optimal performance and energy on multicore/multithreaded processors. The authors demonstrate that the load balancing approach used in state-of-the-art operating systems eliminates one of the big advantages of this architecture: the ability to use unbalanced schedules to allocate the right amount of execution resources to each thread.

Other studies analyze thread scheduling for platforms comprised of several multithreaded processors [38], [54]. McGregor et al. [38] 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. [54] 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. [33] and Shelepov et al. [49] 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.

Other studies propose solutions for thread assignment of multithreaded network workloads in parallel processors, specifically in network processors. Kokku et al. [32] 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 thread re-allocation because of network traffic variations, and provide thread assignment solutions based on application profiling and traffic analysis.

Our previous paper [44] is the first publication to apply Extreme Value Theory to the thread assignment problem. We extend this work in the current paper, in three directions. First, we present a detailed analysis of the technique's prerequisites, including their validation using statistical tests. Second, in Section 5, we propose and evaluate the Sample Pruning POT (SP-POT) method. Third, in Section 6, we evaluate the effect of the sample size on the random sampling approach, concluding that the method should only be used when the random sample contains at least several hundred or several thousand assignments.

An interesting avenue for future work would be to analyze whether random sampling could be used for heterogeneous architectures, e.g. network processors with a number of special-purpose accelerators. In these architectures, the process scheduling problem is tightly coupled with the assignment of sections of code to processing engines, whether general purpose cores or accelerators. One of the main challenges would be to design a sampling method that properly covers the exploration space of all possible process schedules.

Optimal performance analysis. To the best of our knowledge, the work of Jiang et al. [29] is the only systematic study of optimal thread assignment to multithreaded processors. The authors analyze the complexity of this problem, and they propose several thread assignment algorithms. The authors use graphs to model the interaction between simultaneously-running tasks, and they use graph search to find the optimal solution. The main drawback of the study is that the impact of thread interaction on system performance is assumed to be known beforehand, for all possible assignments, which is not generally the case.

We present a different approach for finding the performance of the optimal thread assignment. We do not try to find the best-performing assignment, but to capture a thread 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 thread assignments.

Extreme Value Theory in other domains of computer science. EVT has also been used in real-time systems for the estimation of the worst-case execution time (WCET) of time-critical applications. In real-time environments, and especially in safety-critical systems, it is indispensable to provide tight and reliable WCET estimation with exceptionally high confidence (e.g. 99.9999999999 percent). One of the most important problems in real-time systems is to ensure that the system conditions during experimentation and system analysis include all conditions that can lead to WCET during system deployment. In order to address this problem, several approaches [11] propose introducing randomization into the timing behavior of the system hardware and software.

8 Conclusions and Long-Term Impact

The introduction of multithreaded processors has made the assignment of running threads to hardware contexts an important aspect of system performance. Optimal thread assignment is an NP-complete problem, and exhaustive search is impractical given the large number of possible thread assignments. Numerous studies have proposed heuristic-based algorithms, but they are seldom evaluated in comparison with the optimal solution and they usually make no guarantees on or estimates of their deviation from optimality, making it hard to know whether to invest additional effort to improve the algorithm.

In this paper, we proposed a statistical approach to the problem of optimal thread assignment. We made three main contributions. First, we presented the POT method, which predicts the performance of the optimal thread assignment based on the performance of each thread assignment in a random sample. The method uses techniques from EVT, a branch of statistics that analyses extreme deviations from the population median. Second, we proposed sample pruning, which predicts the performance of each thread assignment in the sample, so that only those with moderate to high predicted performance need to be executed on the target hardware platform. We use sample pruning in the SP-POT method, reducing the analysis time from about 2 hours to about 15 minutes introducing negligible estimation error. Finally, we showed that the performance of the best thread assignment in a random sample is likely to be close to optimal. When a heuristic-based approach is not available, a good thread assignment can be found using random sampling on its own, with the POT or SP-POT method estimating its deviation from optimality.

Many other problems in computer science are NP-complete [22], and therefore intractable. Examples are found in network design, program optimization, data storage and retrieval, process scheduling, graph and automata theory. Since problems typically have a vast exploration space, it is infeasible to find the optimum using an exhaustive search. Intractable problems in computer science are usually addressed using a heuristics-based approach, designed for a specific problem and metric. Design of heuristics-based approaches and tuning for different problems and metrics requires significant effort, and it requires a deep understanding of the application and the target.

The methods presented in this study are independent of the problem that is being addressed. They can therefore be applied to different intractable problems and metrics. They do not require a profound understanding of the target system, which significantly reduces the investment in effort and time. In fact, we have already successfully applied these techniques to compilation of multithreaded streaming applications [43]. The approach is also independent of the target metric, and it can analyze a given system multiple times using different metrics. It is therefore particularly well suited for systems that require analysis of different metrics such as performance, fairness, hardware utilization, energy or power consumption.

EVT is an important branch of statistics, which has found multiple applications in civil engineering, material testing, finance and risk management. It provides many powerful theorems and tools, which could be of great interest to the computer science and engineering community. Its application in these fields is, however, currently marginal. We believe that communicating this paper to a broader audience will encourage new and fruitful applications of EVT in diverse fields of computer science and engineering.

Acknowledgments

This work has been supported by the Ministry of Economy and Competitiveness of Spain (contract TIN-2012-34557), Generalitat de Catalunya (2014-SGR-1272 and 2014-SGR-1051), Severo Ochoa Program (SEV-2011-00067) of the Spanish Government, and by the European Research Council under the European Union's 7th FP, ERC Grant Agreement (number 321253). Miquel Moreto has been partially supported by the Ministry of Economy and Competitiveness under Juan de la Cierva postdoctoral fellowship (JCI-2012-15047).

References

[1] *OpenSPARC T2 System-On-Chip (SOC) Microarchitecture Specification*. Sun Microsystems, Inc, 2007.

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

[3] J. Aas, "Understanding the Linux 2.6.8.1 CPU scheduler," *Silicon Graphics, Inc. (SGI)*, 2005.

[4] C. Acosta, F. Cazorla, A. Ramirez, and M. Valero, "Thread to core assignment in SMT on-chip multiprocessors," in *Proc. 21st Int. Symp. Comput. Archit. High Performance Comput. (SBAC-PAD)*, 2009.

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

[6] A. A. Balkema and L. de Haan, "Residual life time at great age," *Ann. Probability*, vol. 2, 1974.

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

[8] J. R. Bulpin and I. A. Pratt, "Hyper-threading aware process scheduling heuristics," in *Proc. USENIX Annu. Tech. Conf. (ATEC)*, 2005.

[9] E. Castillo and A. Hadi, "Fitting the generalized pareto distribution to data," *J. American Statist. Assoc.*, vol. 92, 1997.

[10] E. Castillo, *Extreme Value Theory in Engineering*. Academic Press, Inc., 1988.

[11] F. Cazorla, T. Vardanega, E. Quinones, and J. Abella, "Upper-bounding program execution time with extreme value theory," in *Proc. WCET workshop*, 2013.

[12] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *Proc. 11th Int. Symp. High-Performance Comput. Archit. (HPCA)*, 2005.

[13] H. Chernoff, "On the distribution of the likelihood ratio," *Ann. Math. Statist.*, vol. 25, 1954.

[14] W. G. Cochran, *Sampling Techniques*, 3rd edition. Wiley-India, 2007.

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

[16] M. De Vuyst, R. Kumar, and D. Tullsen, "Exploiting unbalanced thread scheduling for energy and performance on a CMP of SMT processors," in *Proc. 20th IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2006.

[17] D. Doucette and A. Fedorova, "Base vectors: A potential technique for microarchitectural classification of applications," in *Proc. Workshop Interaction between Oper. Syst. Comput. Archit. (WIOSCA)*, 2007.

[18] A. El-Moursy, R. Garg, D. Albonesi, and S. Dwarkadas, "Compatible phase co-scheduling on a CMP of multi-threaded processors," in *Proc. 20th IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2006.

[19] S. Eyerman and L. Eeckhout, "Per-thread cycle accounting in SMT processors," in *Proc. 14th Int. Conf. Archit. Support Programm. Lang. Oper. Syst. (ASPLOS)*, 2009.

[20] S. Eyerman and L. Eeckhout, "Probabilistic job symbiosis modeling for SMT processor scheduling," in *Proc. 15th Int. Conf. Archit. Support Programm. Lang. Oper. Syst. (ASPLOS)*, 2010.

[21] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum, "Performance of multithreaded chip multiprocessors and implications for operating system design," in *Proc. Ann. Conf. USENIX Annu. Tech. Conf. (ATEC)*, 2005.

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

[23] M. Gilli and E. Këllezi, "An application of extreme value theory for measuring financial risk," *Comput. Econ.*, vol. 27, 2006.

[24] R. Gioiosa, F. Petrini, K. Davis, and F. Lebaillif-Delamare, "Analysis of system overhead on parallel computers," in *Proc. 4th IEEE Int. Symp. Signal Process. Inf. Tech. (ISSPIT)*, 2004.

[25] D. Griffin and A. Burns, "Realism in statistical analysis of worst case execution times," in *Proc. 10th Int. Workshop Worst-Case Execution Time Anal.*, Jul. 2010.

[26] S. Grimshaw, "Computing the maximum likelihood estimates for the generalized Pareto distribution to data," *Technometrics*, vol. 35, 1993.

[27] J. R. M. Hosking and J. R. Wallis, "Parameter and quantile estimation for the generalised Pareto distribution," *Technometrics*, vol. 29, 1987.

[28] J. Jahn, S. Pagani, S. Kobbe, J.-J. Chen, and J. Henkel, "Optimizations for configuring and mapping software pipelines in many core systems," in *Proc. 50th Annu. Design Autom. Conf. (DAC)*, 2013.

[29] Y. Jiang, X. Shen, J. Chen, and R. Tripathi, "Analysis and approximation of optimal co-scheduling on chip multiprocessors," in *Proc. 17th Intl. Conf. Parallel Archit. Compilation Tech. (PACT)*, 2008.

[30] E. Këllezi and M. Gilli, "Extreme value theory for tail-related risk measures," International Center for Financial Asset Management and Engineering, FAME Research Paper Series, 2000.

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

[32] 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.*, vol. 34, no. 1, 2004.

[33] 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 *Proc. 31st Int. Symp. Comput. Archit. (ISCA)*, 2004.

[34] Y.-K. Kwok and I. Ahmad, "Benchmarking and comparison of the task graph scheduling algorithms," *J. Parallel Distrib. Comput.*, vol. 59, no. 3, Dec. 1999.

[35] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Comput. Surveys*, vol. 31, no. 4, Dec. 1999.

[36] C. Luque, M. Moreto, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, and M. Valero, "CPU accounting in CMP processors," *IEEE Comput. Archit. Lett.*, vol. 8, no. 1, Jan.-Jun. 2013.

[37] C. Luque, M. Moreto, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, and M. Valero, "ITCA: Inter-task conflict-aware CPU accounting for CMPs," in *Proc. 18th Int. Conf. Parallel Archit. Compilation Tech. (PACT)*, 2009.

[38] R. L. McGregor, C. D. Antonopoulos, and D. S. Nikolopoulos, "Scheduling algorithms for effective thread pairing on hybrid multiprocessors," in *Proc. 19th IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2005.

[39] S. Parekh, S. Eggers, H. Levy, and J. Lo, "Thread-sensitive scheduling for SMT processors," Tech. Rep. 2000-04-02, Dept. Comput. Sci. Eng., Univ. Washington, 2000.

[40] F. Petrini, D. J. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q," in *Proc. 2003 ACM/IEEE Conf. Supercomput. (SC)*, 2003.

[41] J. I. Pickands, "Statistical inference using extreme value order statistics," *Ann. Statist.*, vol. 3, 1975.

[42] P. Radojković, V. Čakarević, J. Verdú, A. Pajuelo, R. Gioiosa, F. J. Cazorla, M. Nemirovsky, and M. Valero, "Measuring operating system overhead on CMT processors," in *Proc. 20th Int. Symp. Comput. Archit. High Performance Comput. (SBAC-PAD)*, 2008.

[43] P. Radojković, P. M. Carpenter, M. Moretó, A. Ramirez, and F. J. Cazorla, "Kernel partitioning of streaming applications: A statistical approach to an np-complete problem," in *Proc. 45th Int. Symp. Microarchitecture (MICRO)*, 2012.

[44] P. Radojković, V. Čakarević, M. Moretó, J. Verdú, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero, "Optimal task assignment in multithreaded processors: A statistical approach," in *Proc. 17th Int. Conf. Archit. Support Programm. Lang. Oper. Syst. (ASPLOS)*, 2012.

[45] 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 *Proc. 15th ACM SIGPLAN Symp. Principles Practice Parallel Programm. (PPoPP)*, 2010.

[46] P. Radojković, V. Čakarević, J. Verdú, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero, "Thread assignment of multithreaded network applications in multicore/multithreaded processors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 12, pp. 2513–2525, Dec. 2013.

[47] M. Roth, M. J. Best, C. Mustard, and A. Fedorova, "Deconstructing the overhead in parallel applications," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, 2012.

[48] A. Settle, J. Kihm, A. Janiszewski, and D. Connors, "Architectural support for enhanced SMT job scheduling," in *Proc. 13th Int. Conf. Parallel Archit. Compilation Tech. (PACT)*, 2004.

[49] 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 *Proc. ACM SIGOPS Oper. Syst. Review*, 2009.

[50] 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 *Proc. 22nd Ann. Int. Conf. Supercomput. (ICS)*, 2008.

[51] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreaded processor," in *Proc. 9th Int. Conf. Archit. Support Programm. Lang. Oper. Syst. (ASPLOS)*, 2000.

[52] A. Snavely, D. M. Tullsen, and G. Voelker, "Symbiotic jobscheduling with priorities for a simultaneous multithreading processor," in *Proc. 2002 ACM SIGMETRICS Int. Conf. Meas. Modeling Comput. Syst.*, 2002.

[53] N. Tajvidi, "Design and implementation of statistical computations for generalized Pareto distributions," Tech. Rep., Chalmers Univ. Technol., Gothenburg, Sweden, 1996.

[54] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: Sharing-aware scheduling on SMP-CMP-SMT multiprocessors," in *Proc. 2nd ACM SIGOPS/EuroSys Eur. Conf. Comput. Syst. EuroSys*, 2007.

[55] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, "The impact of memory subsystem resource sharing on datacenter applications," in *Proc. 38th Int. Symp. Comput. Archit. (ISCA)*, 2011.

[56] L. A. Torrey, J. Coleman, and B. P. Miller, "A comparison of interactivity in the Linux 2.6 scheduler and an MLFQ scheduler," *Softw. - Practice Exp.*, vol. 37, no. 4, 2007.

[57] V. Čakarević, P. Radojković, J. Verdú, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero, "Characterizing the resource-sharing levels in the UltraSPARC T2 processor," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2009.

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

[59] L. Weng, C. Liu, and J.-L. Gaudiot, "Scheduling optimization in multicore multithreaded microprocessors through dynamic modeling," in *Proc. ACM Int. Conf. Comput. Frontiers (CF)*, 2013.

[60] S. S. Wilks, "The large-sample distribution of the likelihood ratio for testing composite hypotheses," *Ann. Math. Statist.*, vol. 9, 1938.

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

[62] T. Wolf, N. Weng, and C.-H. Tai, "Design considerations for network processor operating systems," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, 2005.

[63] D. Zhan, H. Jiang, and S. C. Seth, "STEM: Spatiotemporal management of capacity for intra-core last level caches," in *Proc. 43rd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2010.

[64] D. Zhan, H. Jiang, and S. C. Seth, "Locality & utility co-optimization for practical capacity management of shared last level caches," in *Proc. 26th ACM Int. Conf. Supercomput. (ICS)*, 2012.

Petar Radojković received the MSc degree in computer science from the University of Belgrade, Belgrade, Serbia, in 2006. He received the MSc degree in computer architecture, networks and systems from the Universitat Politècnica de Catalunya, Barcelona, Spain, in 2009, where he also received the PhD degree from the Department of Computer Architecture in 2013. He is currently a senior researcher at the Barcelona Supercomputing Center, Barcelona, Spain.

Paul M. Carpenter received the Graduation degree from the University of Cambridge, Cambridge, United Kingdom, in 1997, and the PhD degree in computer architecture from the Universitat Politècnica de Catalunya, Barcelona, Spain, in 2011. He is currently a senior researcher at the Barcelona Supercomputing Center (BSC), Barcelona, Spain. Prior to starting his Ph.D., he was senior software engineer at ARM in Cambridge.

Miquel Moretó received the BSc, MSc, and PhD degrees from the Universitat Politècnica de Catalunya (UPC), Barcelona, Spain. He is currently a senior researcher at the Barcelona Supercomputing Center (BSC), Barcelona. Prior to joining BSC, he spent 15 months as a postdoctoral fellow at the International Computer Science Institute (ICSI), Berkeley, CA. His research interests include studying shared resources in multithreaded architectures and hardware-software codesign for future massively parallel systems.

Vladimir Čakarević received the MSc degree in electrical engineering from University of Belgrade, Belgrade, Serbia, in 2006, and the MSc degree in computer architecture, networks and systems from Universitat Politècnica de Catalunya (UPC), Barcelona, Spain, in 2008. He is currently working toward the PhD degree in the Department of Computer Architecture at the Universitat Politècnica de Catalunya. He is affiliated with Barcelona Supercomputing Center.

Javier Verdú received the PhD degree from the Universitat Politècnica de Catalunya (UPC), Barcelona, Spain, in 2008. He is currently a tenure-track lecturer in the Department of Computer Architecture at the Universitat Politècnica de Catalunya. His research interests include performance analysis, optimization of hardware and software multithreading, and cloud based systems.

Alex Pajuelo received the MSc degree in computer science and the PhD degree from the Universitat Politècnica de Catalunya (UPC), Barcelona, Spain, in 1999 and 2005, respectively. He is currently an associate professor in the Department of Computer Architecture at the UPC. His research interests include performance evaluation methodologies, dynamic binary optimization and complex computing-demanding 3D visualization applications.

Francisco J. Cazorla is currently a researcher in the National Spanish Research Council and Barcelona Supercomputing Center (BSC), Barcelona, Spain. He is currently the leader of the group on Interaction between the Operating System and the Computer Architecture at BSC (www.bsc.es/caos). He has worked in research projects with several processor vendor companies (Intel, IBM, Sun Microsystems), as well as in European FP6 and FP7 Projects.

Mario Nemirovsky is currently an ICREA research professor at the Barcelona Supercomputing Center, Barcelona, Spain. He was an adjunct professor at the University of California at Santa Barbara, CA, from 1991 to 1998. He has done research in many areas of computer architecture, including simultaneous multithreading, high-performance architectures, BigData, IoT, FOG Computing, real-time systems and network processors.

Mateo Valero is currently a full professor at Department of Computer Architecture, Universitat Politècnica de Catalunya and the director of Barcelona Supercomputing Center, Barcelona, Spain. He published 600 papers and served in organization of 300 international conferences. His main awards are: Eckert-Mauchly, Harry Goode, ACM Distinguished Service, “Hall of Fame” member IST European Program, King Jaime I in research, two Spanish National Awards on Informatics and Engineering. Honorary Doctorate: Universities of Chalmers, Belgrade, Las Palmas, Zaragoza, Complutense of Madrid and University of Veracruz. He is a Fellow of IEEE, ACM, and Intel Distinguished Research Fellow. He is a member of Royal Spanish Academy of Engineering, Royal Academy of Science and Arts, correspondent academic of Royal Spanish Academy of Sciences, Academia Europaea and Mexican Academy of Science.