Improving the Effective use of Multithreaded Architectures: Implications on Compilation, Thread Assignment, and Timing Analysis

Petar Radojković

Barcelona, 2013

A thesis submitted in fulfillment of
the requirements for the degree of

DOCTOR OF PHILOSOPHY / DOCTOR PER LA UPC

Department of Computer Architecture
Technical University of Catalonia

Abstract

Currently, multithreaded architectures are the mainstream in the processor design. They are widely used in servers, desktop computers, lap-tops, and mobile devices. Multithreaded processors are a very good solution for exploiting processor performance beyond the limitations imposed by limited instruction-level parallelism and power wall. They deliver superior performance and higher performance-per-watt ratio than single-threaded architectures for a given target throughput.

On the other hand, multithreaded architectures introduce several challenges at the software level. In order to utilize the hardware potential optimally and to deliver maximum performance, state-of-the-art multithreaded architectures have to execute numerous software threads simultaneously. At the operating system level, one of the main challenges becomes how to schedule simultaneously-running software threads. Also, the need for numerous software threads motivates the development of multithreaded applications and algorithms. Development of efficient, portable, and correct multithreaded software requires novel programming models and paradigms, and increases the complexity of compilers. In time-critical environments (real-time systems), the timing analysis has to estimate the impact of the collision in shared hardware resources between simultaneously-running threads on the execution time of each thread.

In this thesis, we present cross-domain approaches that improve the effective use of multithreaded architectures. First, we demonstrate the importance of scheduling of network applications running in state-of-the-art multithreaded servers, and propose several methods that improve it. We further analyze optimizations that improve the compilation of multithreaded applications. In particular, we analyze the problem of graph partitioning that is a part of the compilation process of multithreaded streaming applications. Finally, we analyze the impact of shared resources in multithreaded processors in time-critical environments. Our study provides a systematic method for measurement-based timing analysis of applications running on multithreaded architectures.

Acknowledgments

The work presented in this thesis was done mostly at Barcelona Supercomputing Center (BSC), Barcelona, Spain. BSC provided financial support and access to resources that were indispensable for our research. Our work was also supported by the Ministry of Science and Innovation of Spain (contract number TIN-2007-60625), HiPEAC European Network of Excellence on High Performance and Embedded Architecture and Compilation, Sun Microsystems Inc. (now Oracle), and Thales Research and Technology.

Petar Radojković holds the FPU grant AP2008-02370 (Programa Nacional de Formación de Profesorado Universitario) of the Ministry of Education of Spain. During his secondary education and undergraduate studies, Petar Radojković received several grants from Ministry of Education of Republic of Serbia.

The work related to evaluation of multithreaded Commercial-Off-The-Shelf processors in time-critical environments was done mostly during the author's internship at Embedded Systems Lab, Thales Research and Technology, Palaiseau, Paris area, France (December 2010 - March 2011). The internship was funded by HiPEAC. In 2008, Petar Radojković spent three months at Netra DPS group, Sun Microsystems Inc., Menlo Park, California, US. The internship was funded by HiPEAC and "Real-time Chip Multithreading systems" project between BSC and Sun Microsystems Inc.

Professional development of the author was monitored and consulted by Grupo Psicoac S.L., Barcelona.

I would like to thank my advisers for their time and patience. This dissertation could not have been written without their help and guidelines.

Personal gratitude to my advisers, and gratitude to colleagues, friends, and family will be expressed in person. This is the only way for me to do it sincerely and without censorship. The ones who helped me to become who I am deserve no less.

Contents

Chapter 1

Introduction

In the past, at the hardware level, two essential techniques were used to improve the processor performance: increasing the operating frequency (frequency scaling) and improving the processor design (mostly at microarchitecture level). Because of improvements in the transistor implementation technology, the number of transistors on the die duplicated in each processor generation [112]. Additional transistors were used mainly to improve single-threaded processor performance. Each new generation of the processors had deeper and more complex pipelines, more complex prediction engines, and larger on-chip cache memories. Although the manufacturing technology continues to improve and to provide significantly larger number of transistors in each new generation, there are two primary obstacles for additional development of single-threaded processors:

- **Power wall:** The power consumption of the processor scales super-linearly with the working frequency (clock rate) [79]. High power consumption causes several problems. First, it is becoming complex to supply the die with the required electrical current (required power). Second, cooling-down of the die becomes more difficult. Power dissipation has reached the limitation of the reliable processor operation. Third, energy consumption is an important portion of the maintenance cost of IT systems.

- **Limited Instruction Level Parallelism (ILP):** Data and control dependencies limit ILP. It is increasingly difficult to find sufficient parallelism in the instructions stream of a single process to use hardware resources of single-threaded processors effectively [79].

Limited ILP and the power wall motivated the design of multithreaded[1] processors. In multithreaded architectures, the transistors provided by the technology improvement are used to build multiple processing engines that can simultaneously execute different software threads. The main idea behind multithreaded processors is to use simpler processing engines working on a lower frequency, to deliver moderate single-threaded, but superior overall performance. Currently, multithreaded architectures are the mainstream in the processor design. They are widely used in servers, desktop computers, lap-tops, and mobile devices.

In addition to ILP, which is still exploited for each software thread independently, multithreaded processors increase the system throughput by exploiting Thread Level Parallelism (TLP). State-of-the-art multithreaded architectures exploit different TLP paradigms. The most commonly used ones are Simultaneous Multithreading (SMT), Fine-Grain Multithreading (FGMT), and Chip-Multiprocessing (CMP). Also, most of the high-end multithreaded processors such as the Sun UltraSPARC T1 [6, 7], T2 [8, 9], and T4 [13], the IBM POWER5 [142], POWER6 [101], and POWER7 [12] and the Intel core i7 [14], combine different TLP paradigms on a singe die. This allows better utilization of the hardware resources and improves the overall system performance.

1.1 Challenges for the effective use of multithreaded processors

Multithreaded processors are a very good solution for exploiting processor performance beyond the limitations imposed by limited ILP and power wall. They deliver superior performance and higher performance-per-watt ratio than single-threaded architectures for a given target throughput [79]. However, on the other hand, multithreaded architectures introduce several challenges at the software level. In order to utilize the hardware potential optimally, and to deliver maximum performance, state-of-the-art multithreaded architectures have to execute numerous software threads simultaneously. At the operating system level, one of the main challenges becomes how to schedule concurrently-running

[1] In this thesis, we will use the term "multithreaded processor" to refer to any processor that can execute more than one thread simultaneously. Chip-Multiprocessors, Simultaneous Multithreading, Coarse-grain Multithreading, Fine-Grain Multithreading processors or any combination of them are multithreaded processors.

software threads [141]. Also, the need for numerous threads[2] motivates the development of multithreaded applications and algorithms. Development of efficient, portable, and correct multithreaded software requires novel programming models and paradigms, and increases the complexity of compilers. In time-critical environments (real-time systems), the timing analysis has to estimate the impact of the collision in shared hardware resources between simultaneously-running tasks on the execution time of the tasks.

In this thesis, we present cross-domain approaches that improve the effective use of multithreaded architectures. First, we demonstrate the importance of scheduling of network applications running in state-of-the-art multithreaded servers, and propose several methods that improve it. We further analyze optimizations that improve the compilation of multithreaded applications. In particular, we analyze the problem of graph partitioning that is a part of the compilation process of multithreaded streaming applications. Finally, we present a method that improves the analysis of the timing behavior or real-time applications executed in multithreaded architectures. The method quantifies the slowdown that simultaneously-running tasks may experience due to collision in shared processor resources.

1.1.1 Process scheduling

In multithreaded processors, concurrently-running (co-running) threads share and compete for hardware resources. The collision of co-running threads in shared processor hardware resources affects the execution time of the threads and the overall system performance. Moreover, state-of-the-art multithreaded processors have different *levels of resource sharing* [154]. For example, in a CMP processor where each core supports the concurrent execution of several threads through SMT, all simultaneously-running threads share global resources such as the last level of cache memory or the I/O. In addition to this, threads running in the same core share resources such as the Instruction Fetch Unit, or the L1 instruction and data cache. Therefore, the way that threads 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, thread scheduling is comprised of two steps (see Figure 1.1). In the first step, usually called *workload selection*, the

[2] In this thesis, we use the term "thread" to refer to a software i.e. application thread. Words "software thread", "application thread", "thread", and "task" are used interchangeably. When we refer to a hardware context of the processor, we also use the terms "hardware thread", "strand", "logical CPU", and "virtual CPU".

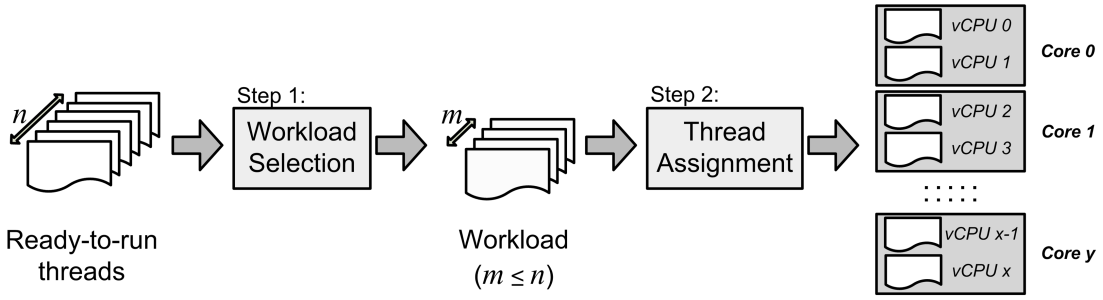


Figure 1.1: Process scheduling in multithreaded processors with several levels of resource sharing: *Workload selection* and *Thread assignment*

operating system (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, vCPU) of the processor.

1.1.1.1 Thread assignment of network applications

In this thesis, we address the problem of thread assignment of network applications running on massive multithreaded processors that can execute tens of software tasks concurrently. The importance of network applications constantly increases with the continuous growth of the Internet traffic and the number and the complexity of the services that these applications provide. In order to provide high-speed processing and high throughput, network applications have specific hardware and operating system requirements.

In networking, the traffic is composed of numerous packets from different users that can be processed at a time. Massively multithreaded processors like UltraSPARC T2 support concurrent execution of multiple threads that can process concurrently the packets with no data dependencies. This makes massively multithreaded processors particularly well suited for the networking systems.

At the operating system level, network-oriented systems require runtime environments that provide high performance, high-speed packet processing, and execution time predictability. In order to address these requirements, network-oriented systems use low-overhead runtime environments that minimize the impact of the system management tasks to the application performance [11]. In networking environments, applications continuously process different packets repeating similar processing algorithm for each packet. As applications running on network processors provide a constant set of services and pro-

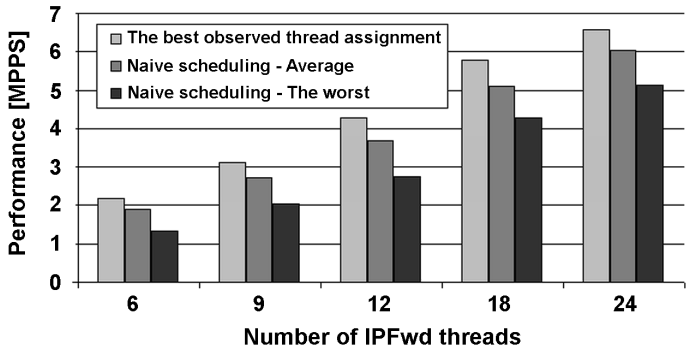


Figure 1.2: Performance of the IPFwd benchmark

cess different packets in a similar manner, a dynamic thread scheduling is not required. In addition to this, the workload running in network processor is usually known beforehand and does not change at runtime. Therefore, the most promising way to improve the performance of network applications running on massive multithreaded processors is to determine a close-to-the-optimal thread assignment, which is the focus of our study.

1.1.1.2 Importance of the thread assignment

In order to illustrate the importance of the thread assignment of network applications, we execute several workloads composed of multithreaded IP Forwarding (IPFwd) application [11] in different thread assignments on the UltraSPARC T2 processor. We executed experiments with 2, 3, 4, 6, and 8 IPFwd instances. Each IPFwd instance is comprised of three threads, Receiving, Processing, and Transmitting, that process the packets in a pipelined manner[3]. Thus, the workload was composed of 6, 9, 12, 18, and 24 IPFwd threads. We could not execute more than 24 IPFwd threads (8 IPFwd instances) simultaneously because of the limitation in the experimental environment. The on-chip Network Interface Unit (NIU) of the UltraSPARC T2 processor used in the study can split the incoming network traffic into up to eight DMA channels, and we could bind at most one receiving thread (at most one IPFwd instance) to each DMA channel.

In each set of the experiments, we executed and measured the performance of between 700 and 1500 different thread assignments (see Section 2.4). The results of the experiments are presented in Figure 1.2. The X-axis of the figure shows the number of threads in the workload, while the Y-axis shows the system performance in processed Millions of Packets Per Second (MPPS).

[3]IPFwd application is described in detail in Section 2.3.

The figure presents three groups of bars. Bar *'The best observed thread assignment'* shows the performance of the best observed thread assignment for each number of IPFwd threads. Bars *'Naive scheduling-Average'* and *'Naive scheduling-The worst'* show the average and the worst performance of the naive thread assignment, the technique that randomly distributes the running threads among the processor cores and hardware pipelines of the processor. The results presented in Figure 1.2 show that thread assignment has a significant impact on the system performance. The average performance loss of naive scheduling with respect to the performance of the detected best-performing thread assignment ranges from 290,000 PPS (six IPFwd threads) to 680,000 PPS (18 threads). The worst detected performance loss of naive tasks assignment ranges up to 1.5 million PPS (12 threads), which represent the degradation of 35%.

By choosing a given thread assignment, we implicitly decide which hardware resources are shared between concurrently-running (co-running) threads. State-of-the-art OSs, like Linux and Solaris, use load balancing mechanism [15][149] to distribute the load of the running processes among all the available logical CPUs (*i.e.* cores in a CMP architecture or hardware contexts in a SMT architecture). Also, cache and TLB affinity algorithms [15][149] try to keep threads in the same logical CPU to reduce cache misses and page faults as much as possible. Although these characteristics improve performance, they do not fully exploit the capabilities of the current multithreaded processors with several levels of resource sharing. The main drawback of the thread assignment algorithms used in current OSs is that they do not consider the hardware requirements of each thread; thus they are not able to provide thread assignments that optimally use processor hardware resources [50]. We analyze real network applications running on state-of-the-art server comprising UltraSPARC T2 processor and measure significant performance loss introduced by Linux-like process scheduler.

1.1.1.3 Intractability of the thread assignment problem

Analytical analysis of optimal thread assignment is an NP-complete problem [67]. Also, in current multithreaded processors comprised of several cores where each core supports several simultaneously-running threads, the total number of possible thread assignments is vast [50, 59, 87, 131]. This number will grow rapidly in future massively multithreaded processors in which the number of cores and number of different levels of resource sharing increase [116]. In order to illustrate this, we study the number of possible assignments when several threads simultaneously execute on the UltraSPARC T2 processor.

Number of threads	Number of possible thread assignments	Time needed to execute all thread assignments	Time needed to predict all thread assignments
3	11	11 seconds	11 µs
6	1,526	25 minutes	1.5 ms
9	591 x 10^3	7 days	0.6 seconds
12	459 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 1.1: Number of different thread assignments for applications running 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 threads at a time, meaning that the maximum number of simultaneously running threads is 64[4]. The number of possible thread assignments for different workload size is shown in Table 1.1. The first column of the table shows the number of threads in the workload. We present results for 3, 6, 9, 12, 15, 18, and 60 threads. The second column presents the number of different thread assignments for a given workload size. The third column of the table shows the time needed to execute all possible thread 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 thread 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 threads assignments as well as the time needed to execute them grows rapidly with the number of threads in the workload. The point where running all assignments is unfeasible is reached very fast. For 9 threads in the workload, the time needed to execute all assignments is 7 days, for 12 threads, the execution time is more than 15 years. For 60-thread workloads that use 94% of hardware contexts of the processor, the time needed to execute all possible thread assignments is 1.75×10^{51} years. The results presented in Table 1.1 clearly show that, in general, running all thread assignments is unfeasible, and that an exhaustive search cannot be used to find the optimal system performance for a given workload.

Several studies (see Section 3.5) propose methods that can predict the performance of different thread assignments of a given workload. However, from Table 1.1, we also see that the prediction of all possible threads assignments is unfeasible. The last column

[4]For more information about the UltraSPARC T2 processor used in the study, refer to Section 2.1.

shows that, for example, for the case of 15-thread workloads, the time needed to predict all assignments is 7 days. For 18 threads, the prediction time is measured in years. Therefore, performance predictors can be used only to determine the close-to-the-optimal thread assignment in a limited sample. However, since the number of possible thread assignments in modern processors is vast, the effectiveness of performance predictors is questionable [50, 59]; 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.

Therefore, the current thread 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 thread assignment for a workload is unknown, the room for improvement of the current thread assignment techniques cannot be determined. Thus, it is difficult to determine the effort needed to invest in the thread assignment process. This may lead to overspending if a good thread assignment algorithm is constantly improved, or sub-optimal performance if a poor-performing algorithm is not enhanced.

In this thesis, we address the problem of thread assignment of multithreaded network applications running on processors with several levels of resource sharing. We present two methods, *TSBSched* and *BlackBox scheduler*, that profile the applications running on a target architecture and predict the performance of different thread assignments. We also present a statistical method based on Extreme Value Theory that predicts the performance of the optimal thread assignment. TSBSched, BlackBox scheduler, and the proposed statistical method are evaluated on a set of multithreaded network applications running on a network server that comprises UltraSPARC T2 processor.

1.1.2 Compilation of multithreaded streaming applications

In order to exploit the hardware potential and deliver maximum performance, state-of-the-art multithreaded architectures have to execute numerous software threads simultaneously. One way to use multithreaded architectures effectively is to execute numerous software threads concurrently. For example, Internet, network, and data-center servers can concurrently execute multiple instances of a single application that process (highly-)independent traffic from different users. However, in various domains (such as desktop and mobile applications) it is not trivial to find numerous independent software threads that can execute concurrently. Actually, one of the greatest challenges in software devel-

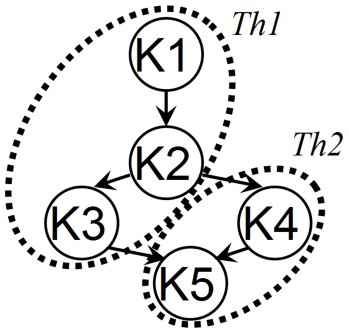


Figure 1.3: A kernel partition example

opment is how to write efficient, portable, and correct software for multithreaded processors. A promising approach to develop multithreaded applications is to expose parallelism to the compiler through the use of domain-specific languages. The compiler can then perform complex transformations that the programmer would otherwise have had to do.

One of the widely-used domain-specific languages is stream programming. Stream programming is suitable for applications that process long sequences of data, such as voice, image, multimedia content, Internet and communication traffic. Stream programming languages [2, 17, 30, 33, 148] represent the program as concurrent kernels, which communicate only via point-to-point streams. A kernel is a basic computation block with a user-defined function that processes one or more input data streams into one or more output data streams. Dependencies between different kernels are described explicitly through the communication data channels. The whole application can be represented as a *stream graph*. The nodes of the stream graph correspond to the kernels, while the directed edges represent the communication data channels. In order to take advantage of multiple processor cores, the stream program is automatically compiled into a multithreaded executable by the stream compiler. One of the most important tasks of the stream compiler is to partition the kernels in the stream graph into software threads. In Figure 1.3, we illustrate a stream graph of a simple program, which is comprised of five kernels (K1 to K5). The figure also shows one possible kernel partition of the graph: kernels K1, K2, and K3 are to be compiled to software thread *Th1*, while kernels K4 and K5 are to be compiled into *Th2*.

Kernel partitioning can affect the overall system performance significantly. For example, for the benchmarks included in the StreamIt 2.1.1 suite [3, 138], the relative performance difference between good and bad kernel partitions of the same benchmark mapped into four software threads ranges from 2.4× to 3.9×, and on average it is 3.5×.

Several studies (see Section 5.6) propose heuristic-based algorithms to address the kernel partitioning problem. As kernel partitioning is an intractable problem [63, 67], it is impossible in general to know the performance of the optimal partition, so the room for improvement is also unknown. It is hard to decide whether to invest additional effort to try to improve a given algorithm, since it may already be close to optimal.

In this thesis, we propose a statistical approach to the kernel partitioning problem. We present a method that predicts the performance of the optimal partition based on the observed performances of a random sample of kernel partitions. We also analyze whether random sampling on its own is likely to find a kernel partition with good performance. The proposed method is evaluated on the kernel partitioning of the benchmarks included in StreamIt 2.1.1 suite.

1.1.3 Multithreaded processors in time-critical environments

Due to good performance-per-watt ratio and high performance opportunities, multithreaded processors are of special interest for embedded real-time systems [152]. These architectures are particularly well suited for embedded *integrated architectures* [22, 113, 158] in which several functions are integrated into the same processor. In this context, multithreaded processors can potentially schedule mixed criticality workloads, *i.e.* workloads composed of safety-critical, mission-critical, and non-critical applications inside the same processor, improving the hardware utilization and so reducing cost, size, weight, and energy consumption [110].

Unfortunately, despite the benefits that multithreaded processors may offer in embedded real-time systems, particularly in integrated architectures, the market has not yet embraced such a shift. In addition to the functional correctness of operations, in real-time systems, it is required to guarantee that the execution time of tasks does not exceed the corresponding *deadlines*. This requisite of real-time systems is referred to as *time correctness*. In order to guarantee time correctness of a given task, it is obligatory to estimate the maximum execution time that the task could have on a specific hardware platform, *i.e.* to determine the *Worst Case Execution Time (WCET)* of the task. In multithreaded processors, simultaneously-running tasks share and compete for processor resources, so the timing analysis has to estimate the possible impact that these inter-task interferences have on the execution time of the applications. As a result, it is much more difficult to provide the WCET estimations for applications running on multithreaded processors than running on single-threaded processors [120].

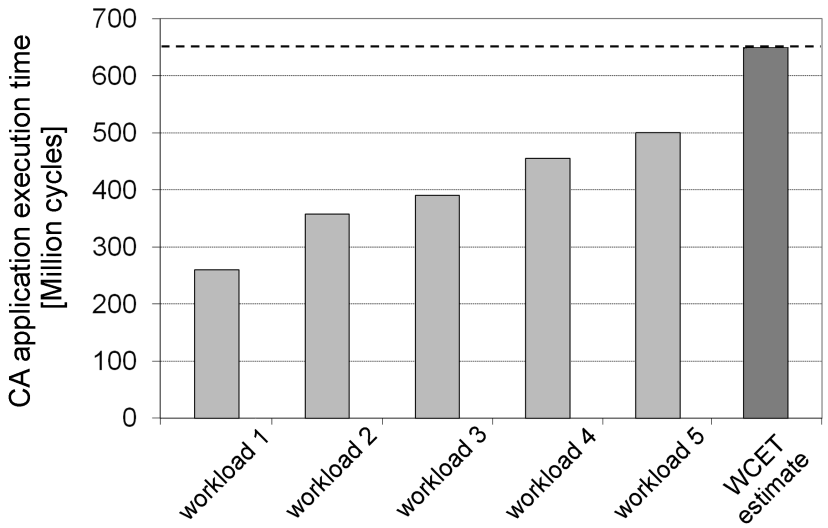


Figure 1.4: Collision Avoidance (CA) application: The execution time in different workloads and the WCET estimate

In addition to this, in order to reduce the cost of the system verification in the time domain, it is required to do an incremental timing verification of real-time tasks. Incremental timing verification means that a user does not have to verify the timing behavior of all concurrently-running applications each time a new component (application) is changed or added to the system [60]. In that sense, the system is time composable if the WCET estimate of the tasks do not change if any of the tasks in the workload change.

In the following example, Paolieri et al. [117] demonstrate the execution time variations of applications running in multithreaded processors, and illustrate the composable WCET estimate for the application under study. In this experiment, the Collision Avoidance (CA) application developed by Honeywell Corporation is executed with different sets of co-running applications (different workloads) on a multicore processor. The CA application is based on 3D path planning algorithm that is used in autonomous-driven airplanes.

The results of the experiment are presented in Figure 1.4. The X-axis of the figure lists different workloads (*workload 1* to *workload 5*) that are executed simultaneously with the CA application. The Y-axis shows the execution time of the CA application in each configuration. From the results presented in Figure 1.4, we see that the execution time of the CA application varies significantly (from 250 to 500 million cycles) depending on the co-running workload. A safe and time-composable WCET estimate for the CA application running on the architecture under study (dark gray bar in Figure 1.4) must be computed independently of the co-running workload, and it must exceed the execution

time of the CA application in *any* possible workload. In this way, the WCET estimate of the CA application is computed only once, and it does not have to be recomputed each time that any of the co-running tasks is replaced or updated.

In this thesis, we present a systematic method for measurement-based timing analysis of applications running on multithreaded architectures, and we demonstrate that the presented method helps with providing composable WCET estimations. We also show how the method can be used to determine whether different multithreaded architectures are suitable for time-critical environments.

1.2 Thesis contributions

This thesis presents cross-domain approaches that improve the effective use of multithreaded architectures. The contributions of the thesis can be classified in three groups. First, we propose several methods for thread assignment of multithreaded network applications running in multithreaded servers. Second, we analyze the problem of graph partitioning that is a part of the compilation process of multithreaded streaming applications. Finally, we present a method that improves the measurement-based timing analysis of multithreaded architectures used in time-critical environments. The following sections briefly summarize each of the contributions.

1.2.1 Thread assignment on multithreaded processors

The first problem that we address in the thesis is thread assignment of multithreaded network applications running on network servers that comprise state-of-the-art multithreaded processors. We propose *TSBSched* (Thread-to-Strand Binding Scheduler) [131], a systematic method for thread assignment of multithreaded network application running on processors with several levels of resource sharing. Based on minimum information about the target processor architecture, and without any data about the hardware requirements of the applications under study, the proposed method determines a set of thread assignments that can be used to model the interference between concurrently running threads (*profiling thread assignments*). The profiling assignments are executed on the processor under study, and the performance of each assignment is measured. Finally, the method uses the measured performance of the profiling thread assignments to estimate the performance of *any* assignment composed of applications under study.

TSBSched is a simple, architecture independent thread assignment approach that determines assignments with a good performance. However, TSBSched has two main limitations: (1) The programmer who profiles the application must have a profound knowledge about the application behavior. (2) Also, TSBSched requires adjustments of the application source code. If the source code is not available, TSBSched method cannot be used.

In order to overcome these limitations, we design *BlackBox* scheduler [132]. BlackBox scheduler does not require inserting any test points in the application source code nor the profound understanding of it. Therefore, BlackBox scheduler can be used when the source code is not available, which is a common limitation.

We also present a statistical approach to the thread assignment problem [130]. We present a method based on Extreme Value Theory (EVT)[26, 34] that predicts of the performance of the optimal thread assignment. Knowing the optimal system performance improves the evaluation of any thread assignment technique and it is the most important piece of information for the system designer when deciding whether any scheduling algorithm should be enhanced. 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* thread assignments of a given workload and measure the performance of each assignment. According to this analysis, there is a 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 multithreaded architectures. Unlike other thread assignment proposals that use performance predictors to find the best thread assignment, the method can be applied directly and without any change to any architecture running any set of applications.

1.2.2 Kernel partitioning of streaming applications

The second problem that is addressed in the thesis is kernel partitioning of streaming applications. We present a method based on EVT that statistically estimates the performance of the optimal kernel partition [128]. To the best of our knowledge, this is the first study that applies EVT to a graph partitioning problem. We use the estimates of the optimal performance to evaluate a state-of-the-art heuristic-based kernel partitioning algorithm. Also, we show that the sampling method used to generate random partitions, has a significant effect on the applicability of the statistical analysis. We analyze different sampling

methods, and our results strongly recommend that the samples should be uniformly distributed. Since a complex heuristic-based algorithm may not always be available, the user may pick the best from a random sample, and measure its quality using the estimates of the optimal performance. We analyze whether random sampling is likely to find a good kernel partition.

1.2.3 Multithreaded processors in time-critical environments

Finally, we analyze the impact of shared resources in multithreaded processors in time-critical environments. Our study provides a systematic method for measurement-based timing analysis of applications running on multithreaded architectures [129]. We define a set of specific resource-stressing benchmarks that introduce a high number of interferences on each potentially-shared hardware resource. By using these *resource-stressing* benchmarks as co-running threads, we obtain a good estimation of the worst-case slowdown that real applications may experience because of collision in shared processor resources.

We also show how the proposed method can be applied to determine the suitability of different multithreaded architectures for time-critical systems. When a workload is composed only of resource-stressing benchmarks, the detected slowdown is unlikely to be exceeded for any workload composed of real applications. Therefore, the slowdown detected when using resource-stressing benchmarks may serve as an upper estimate of the effect of inter-task interference for a given processor. This analysis is important when companies have to determine which architectures are good candidates to be used in their future time-critical systems.

1.3 Thesis structure

The rest of the thesis is organized as follows:

- Chapter 2 presents the experimental environment used to analyze the thread assignment problem. In this chapter, we describe the hardware platform, operating system, and benchmarks used in the study.
- In Chapter 3, we propose and evaluate two systematic methods for thread assignment of multithreaded network applications running on processors with several levels of resource sharing, *TSBSched* and *BlackBox* scheduler.

- Chapter 4 presents a statistical method based on Extreme Value Theory that predicts the performance of the optimal thread assignment in multithreaded processors. We further show that executing a sample of several hundred or several thousand random thread assignments is enough to obtain, with very high confidence, an assignment with a performance that is close to the optimal one.

- Chapter 5 summarizes our work on the problem of kernel partitioning for streaming applications. In this chapter, we present a method that statistically estimates the performance of the optimal kernel partition. We further use the presented method to evaluate a recently-published partitioning algorithm based on a heuristic.

- In Chapter 6, we propose a method that quantifies the slowdown that simultaneously-running tasks may experience due to collision in shared processor resources. This information is a base for incremental timing verification of multithreaded architectures that are used in time-critical environments.

- Chapter 7 lists the conclusions of the thesis, briefly summarizes future work, and lists the relevant publications and awards.

Chapter 2
Experimental setup

This chapter presents the experimental environment that was used to analyze the problem of thread assignment on multithreaded processors. The experimental setup in which we analyzed kernel partitioning for streaming applications is presented in Chapter 5. Chapter 6 includes the description of the environment in which we executed the experiments related to the study of timing behavior of real-time applications running in multithreaded architectures.

We evaluated the proposed thread assignment methods for a set of multithreaded network applications running in a real industrial environment. The environment comprised two Oracle T5220 servers that managed the generation and the processing of network traffic. Each T5220 server comprised one UltraSPARC T2 processor. One T5220 server executed the Network Traffic Generator (NTGen) [11]. NTGen is a software tool, developed by Oracle, that generates IPv4 TCP/UDP packets with configurable options to modify various packet header fields. In addition to this, we enhanced the NTGen to transmit network traffic read from trace files, such as anonymized Internet trace files [5]. T5220 server running NTGen transmitted network packets through a 10Gb link to the second T5220 server in which we executed the thread assignments selected by our method. In all the experiments presented in the study, NTGen generated enough traffic to saturate the network processing server. Thus, in all the experiments, the performance bottleneck was the speed at which the packets were processed, which is determined by the performance of the selected thread assignment.

In order to avoid interferences between user applications and operating system processes, we executed the experiments in Netra DPS, a low-overhead environment used in network processing systems [10, 11]. We briefly describe Netra DPS being focused on the difference between this environment and a fully-fledged operating systems, such as Linux or Solaris. At the end, we present the set of multithreaded network applications

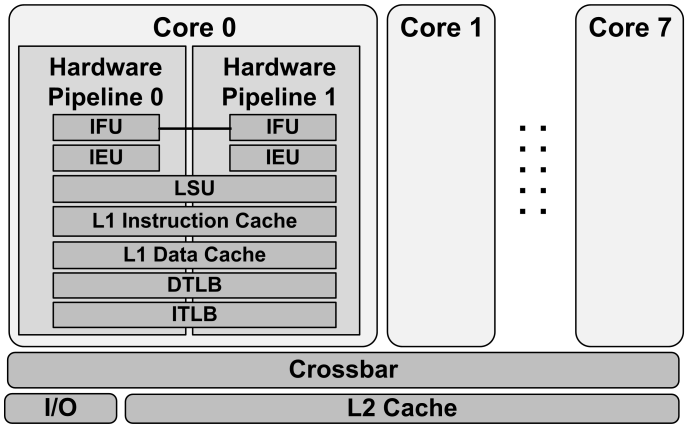


Figure 2.1: Schematic view of the three resource sharing levels of the UltraSPARC T2 processor

and the methodology we use to evaluate the proposed thread assignment approaches.

2.1 UltraSPARC T2 processor

The UltraSPARC T2 is a multithreaded processor [8][9] that comprises eight cores connected through the crossbar to the shared L2 cache (see Figure 2.1). Each of the cores supports eight hardware contexts (strands), thus up to 64 threads 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, from now on also referred as *hardware pipelines* or *hardware pipes*.

Processes simultaneously running on the UltraSPARC T2 processor share (and compete for) different resources depending on how they are assignmed to hardware contexts. As it is shown in Figure 2.1, the resources of the processor are shared on three different levels: *IntraPipe*, between threads running in the same hardware pipeline; *IntraCore*, between threads running in different hardware pipelines of the same core; and *InterCore*, between threads running on different cores [154].

IntraPipe: Resources shared at this level are the Instruction Fetch Unit (IFU) and the Integer Execution Units (IEU). Even if the IFU is physically shared between all processes that execute on the same hardware core, the instruction fetch policy prevents any interaction in the IFU between threads running in different hardware pipelines. That is, the IFU actually behaves as two private IFUs, one for each hardware pipeline.

IntraCore: Threads that execute on the same core share the IntraCore resources: the

16KB L1 instruction cache, the 8KB L1 data cache (2 cycles access time), the Load Store Unit (LSU), the Floating Point and Graphic Unit (FPU), and the Cryptographic Processing Unit.

InterCore: Finally, the main InterCore resources (globally-shared resources) of the UltraSPARC T2 processor are: the L2 cache, the on-chip interconnection network (crossbar), the memory controllers, and the interface to the off-chip resources (such as the I/O). The 4MB 16-way associative L2 cache is partitioned eight banks that operate independently from each other. The L2 cache access time is 22 cycles, and the L2 miss that accesses the main memory lasts around 185 CPU cycles. The L2 cache connects to four on-chip DRAM controllers, which directly interface to a pair of fully buffered memory channels.

In order to fully utilize the performance of a multithreaded processors like the UltraSPARC T2, it is important to understand which hardware resources are shared on each resource-sharing level. In the UltraSPARC T2, two threads running in the same hardware pipeline conflict at all resource-sharing levels: IntraPipe, IntraCore, and InterCore. Threads running in different hardware pipelines of the same core conflict at the IntraCore and InterCore levels. Finally, threads in different cores interact only at InterCore level. On the other hand, threads running in the same processor core communicate through L1 cache shared at the IntraCore level, while threads from different cores share data and instructions only at globally shared L2 cache that has significantly higher access time.

2.2 Netra DPS

State-of-the-art operating systems (OSs) provide features like the dynamic process scheduler or the virtual memory to enhance the execution of user applications. On the other hand, user applications experience slowdown and have variable execution time because of interference with OS processes. Many networking systems use lightweight runtime environments in order to reduce the overhead introduced by fully-fledged OSs. One of these environments is Netra DPS [10, 11] developed by Oracle. Netra DPS provides less functionality than fully-fledged OSs, but also introduces less overhead. In our previous study [133], we compared the overhead introduced by Linux, Solaris, and Netra DPS in the execution of benchmarks running on the UltraSPARC T1 processor. Presented results show that Netra DPS is the environment that clearly exhibits the best and most stable application execution time.

Netra DPS does not incorporate virtual memory nor a run-time process scheduler, and performs no context switching. The assignment of running threads 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 thread will be executed. Netra DPS does not provide any interrupt handler nor daemons. A given thread runs to completion on the assigned hardware context without any interruption.

2.3 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 adapted in order to execute in this environment.

2.3.1 Overview

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, the IPFwd application may have significantly different memory behavior [82, 95]. In order to cover different cases of IPFwd memory behavior, we created three variants of the IPFwd application that are based on the IPFwd application included in the Netra DPS distribution [10]:

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

(2) The table does not fit in the L1 data cache, but it fits in the L2 cache (*IPFwd-L2*). The routing table entries of IPFwd-L2 are configured to cause a lot of data L1 misses in IPFwd traffic processing.

(3) The table does not fit in the L2 cache and the lookup table entries are initialized to make IPFwd continuously access the main memory (*IPFwd-Mem*).

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 [140].

(2) Complex packet processing: Based on the IP Forwarding application included

in the Netra DPS distribution, we also designed the benchmarks that emulate more complex packet processing and multiple accesses to the memory. In these benchmarks, the hash function call and the hash table lookup are repeated N times (three times in our experiments). We refer to these benchmarks as *hashN*. Again, we use three different configurations to analyze complementary scenarios that cover from the best to the worst case studies:

(1) *hashN-L1*: the lookup table fits in the L1 data cache;

(2) *hashN-L2*: the table does not fit in the L1 data cache, but it fits in the L2 cache;

(3) *hashN-Mem*: the table does not fit in the L2 cache.

(3) CPU intensive packet processing: The behavior of CPU-intensive network applications (such as high layer packet decoding or URL decoding) is emulated by the benchmarks that have a high rate of resource conflicts in the IntraPipe and IntraCore processor resources.

In order to stress IntraPipe processor resources, we developed *IPFwd-intadd* benchmark. Benchmark *IPFwd-intadd* is a modification of the IPFwd application included in the Netra DPS. We develop *IPFwd-intadd* by inserting a set of *integer addition* instructions at the end of the IPFwd processing stage. These instructions put significant stress to Instruction Fetch Unit and Integer Execution Unit, both shared between threads running on the same hardware pipeline.

In order to stress IntraCore processor resources, we developed benchmarks *IPFwd-intmul* and *IPFwd-intdiv*. We develop these benchmarks by inserting insert a set of *integer multiplication* and *integer division* instructions at the end of the IPFwd processing stage. In the UltraSPARC T2 processor, *integer multiplication* and *integer division* instructions are executed in the Floating Point Unit (FPU) that is shared between threads executing on the same processor core (IntraCore resource). Benchmarks *IPFwd-intmul* and *IPFwd-intdiv* stress FPU in two different ways [154]: (1) *Integer multiplication* instructions are pipelined with throughput of one instruction per cycle. This means that several *IPFwd-intmul* benchmarks may execute simultaneously in the same processor core and share the FPU with almost no overhead. (2) On the other hand, *integer division* instructions are non-pipelined. When several *integer division* instructions are ready to be executed at the same time, they are serialized – only one instruction is executed at a time, while others are delayed. This may cause significant overhead when several *IPFwd-intdiv* processing threads run on the same processor core.

(4) Packet analyzer is a program that can intercept and log traffic passing over a

network or part of a network [45]. Packet analyzers 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 [4, 145, 163].

The packet analyzer that we used in the experiments captures each packet that passes through the Network Interface Unit (NIU), inspects the packet, and analyzes its content according to the appropriate RFC specifications [84]. 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. 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.

(5) Aho-Corasick is a string matching algorithm. String matching is the basic technique to analyze the network traffic at the application layer [76]. 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 [19]. 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 [114].

In the presented experiments, 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.

(6) Stateful packet processing is an important component of state-of-the art network monitoring tools [115, 156] and intrusion prevention and detection system [145]. Unlike stateless applications that process each packet independently (like the other benchmarks used in our 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-5-tuple* [1], share the common information called the *flow-record* [56]. 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 [56, 115]. The hash table contains 2^{16} entries, which is sufficient to store the records of active flows of fully-utilized 10Gb link [155], and it is the hash table size that was already used to test different network monitoring tools [56].

2.3.2 Implementation

Each benchmark is divided into three threads that form a software pipeline (see Figure 2.2). This is a commonly-used approach in the development of the network applications [11, 164]:

- 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

[1] The flow-5-tuple is consisted of the source and destination IP address, the source and destination port, and the protocol used.

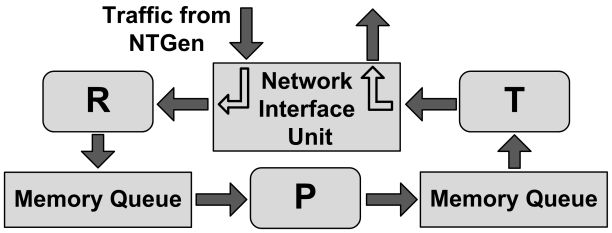


Figure 2.2: The schematic view of the benchmarks

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 threads assignment techniques because:

(1) Each benchmark is divided in three different threads, thus the systems deal with heterogeneous threads 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 [154].

(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 between processor cores (L1 cache domains).

(4) The impact of thread assignment to the performance is significant. We detect performance variation of up to 60% between different thread assignments of the same workload.

2.4 Methodology

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

Chapter 3

Thread assignment of multithreaded network applications on multicore/multithreaded processors

In this chapter, we propose two systematic methods for thread assignment of multithreaded network applications running on processors with several levels of resource sharing, *TSBSched* and *BlackBox* scheduler. We evaluated the proposed methods for a set of multithreaded network applications running on the UltraSPARC T2 processor. The proposed thread assignment methods improve the system performance up to 48% with respect to state-of-the-art scheduling algorithms, and up to 60% with respect to a naive scheduling.

3.1 Introduction

In processors with several levels of resource sharing, thread scheduling comprises two steps: *workload selection* and *thread assignment*. We propose *TSBSched* and *BlackBox scheduler*, systematic methods for thread assignment of multithreaded network application running on processors with several levels of resource sharing. Based on minimum information about the target processor architecture, and without any data about the hardware requirements of applications under study, TSBSched and BlackBox scheduler determine a set of thread assignments that can be used to model the interference between concurrently running threads (*profiling thread assignments*). The profiling assignments are executed on the processor under study and the performance of each assignment is measured. Finally, the methods use the measured performance of the profiling thread assignments to estimate the performance of *any* assignment composed of the applications

under study.

The proposed thread assignment methods are evaluated with an industrial case study for a set of multithreaded networking applications running on the UltraSPARC T2 processor. The results show that the methods provide thread assignments with performance close to the optimal one. In most of the experiments, the proposed thread assignment methods detected the best actual (measured) thread assignment in the evaluation sample. The highest performance difference between the thread assignment provided by TSBSched and BlackBox scheduler and the actual best thread assignments in the evaluation sample is only 3%. The proposed methods also provide a significant performance improvement with respect to the state-of-the-art thread assignment techniques.

The rest of the chapter is organized as follows. Section 3.2 discusses the hardware and OS requirements of network applications, demonstrates the importance of the thread assignment for network applications, and presents the overview of the state-of-the-art thread assignment approaches. Section 3.3 describes in detail the thread assignment methods that we propose. The results of the experiments used in the evaluation of TSBSched and BlackBox scheduler are presented in Section 3.4. Section 3.5 shows the overview of the related work, while Section 3.6 summarizes the study.

3.2 Background

Optimal thread assignment requires a profound knowledge of the resource requirements of each running thread, and the understanding of how the threads interact in each of the shared processor resource. Therefore, it is difficult to determine the optimal thread assignment without 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 concurrently running threads. The problem becomes even more complex for multithreaded applications. When an application is comprised of several threads, it is not sufficient to understand how sharing of hardware resources affects the execution time of each of the threads independently. The designer also has to be aware of the inter-thread communication, and to understand which application thread is the bottleneck that determines the application performance. Currently, thread assignment is done in one of three ways:

(1) **Resource-oblivious** (naive) thread assignment randomly distributes the workload among hardware contexts of the processor. Naive thread assignment provides non-optimal

performance. In our experiments, naive scheduling introduces the performance loss of up to 60% with respect to the optimal thread assignment.

(2) **Manual assignment:** A skilled designer attempts to determine a good thread 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 concurrently running threads. In addition to this, any change in the application or in the hardware platform requires the repetition of the analysis. Manual thread assignment is not a systematic approach, its performance depends on the programmer skills, and, in general, does not provide the thread assignment with the optimal performance.

(3) **Load balancing and cache affinity:** State-of-the-art fully-fledged OSs, like Linux and Solaris, use load balancing mechanism [15][149] to equally distribute the load of the running threads among all the available scheduling domains (e.g. cores of the CMP architecture). In addition to this, cache and TLB affinity algorithms [15][149] keep the threads assigned to the same logical CPU in order to reduce cache and TLB misses caused by the thread migration. Although these techniques improve the performance, they are not sufficient to fully exploit the capabilities of the current multithreaded processors with several levels of resource sharing. De Vuyst et al. [50] demonstrate that the load balancing used in the state-of-the-art operating systems eliminates the unbalanced schedules that allocate the proper amount of processor resources for each of the co-running threads, which is one of the important advantages of the modern multithreaded processors. We analyze real network applications running on the UltraSPARC T2 processor, and measure the performance difference of up to 48% between load-balanced and optimal thread assignments.

3.3 Thread assignment methods

We present *TSBSched* and *BlackBox scheduler*, two systematic methods for thread assignment of multithreaded network applications running on processors with several levels of resource sharing. The purpose of these methods is to estimate the performance of different (many) thread assignments and to determine the assignments that provide a good performance.

Without any data about the hardware requirements of the applications under study, and using the minimum information about the processor under study, the methods determine a set of thread assignments (*profiling thread assignments*) that are used to model the in-

terference between co-running threads. Based on the measured performance of profiling thread assignments, the method estimates the performance of *any* assignment composed of the applications under study.

Thread assignment methods are designed to accomplish two main objectives:

(1) Remove the need for detailed knowledge about the hardware requirements of applications under study: In order to select a good assignment, scheduling methods have to be aware of the interaction between concurrently running threads. Modeling application interference in shared processor resources is a challenging task. Most of the studies that address this problem (see Section 3.5) profile each thread independently, and then predict the performance when several threads execute concurrently on a processor. We measure directly the interaction between concurrently running threads for a limited set of thread assignments, and use this data to model the interference in hardware resources between co-running threads in any given assignment. The main benefit of this approach is that the information about the application hardware requirements is not incorporated into the thread assignment method, but it is encapsulated into the data passed to the method (the profiling data).

(2) Architecture independence: The only architectural data that the methods require is the hierarchy of different levels of resource sharing and the number of hardware contexts (virtual CPUs) in each of them. For example, for the UltraSPARC T2 processor, the architecure description contains the information that: (1) The processor resources are shared in three different levels (IntraPipe, IntraCore, and InterCore); (2) The processor contains eight cores, each of them contains two hardware pipelines, and each hardware pipeline has support for four concurrently running threads. The methods do not require any information about which hardware resources are shared on each level, nor the microarchitecture details of the processor resources (e.g. the size of the cache memory, the number and the characteristics of the execution units, etc.). This is the main reason why the application of the method to different processor architectures requires minimum adjustments.

Figure 3.1 presents the schematic view of the proposed thread assignments methods. The methods are comprised of three phases: (1) Application profiling, (2) Performance prediction, and (3) Selection phase.

(1) **Application profiling phase:** We profile the applications under study. The output of this phase are the *Base Time Table* and *Slowdown Table*. These tables contain all the information that is needed to predict the performance of any thread assignment comprised

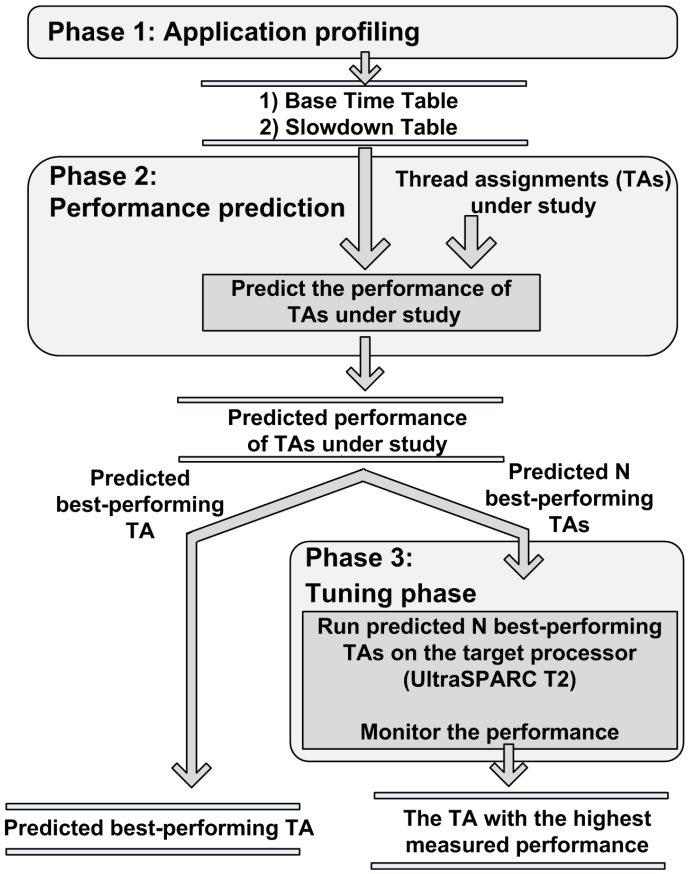


Figure 3.1: Schematic view of TSBSched and BlackBox scheduler

of the applications under study.

(2) **Performance prediction phase:** The models predicts the performance for different thread assignments based on the data stored in the Base Time Table and Slowdown Table.

(3) **Selection phase**: This is an optional step of the algorithm. In the Selection phase, the predicted N best-performing thread assignments (e.g. 5 or 10 assignments) are executed on the target processor and the assignment with the highest measured performance is selected as the final outcome of the method. If the predicted best-performing assignment is not the actual best one, the selection phase can improve the performance of the thread assignment that is the outcome of the method.

The following sections describe each phase of the TSBSched and BlackBox scheduler in detail.

3.3.1 TSBSched

TSBSched (Thread to Strand Binding Scheduler) [131] models the interference between concurrently running threads, by measuring the execution time of each thread running in different assignments on a real hardware. The method compares the execution time of a thread when it runs alone on the processor and when it is co-scheduled with other threads, and uses this data as the input to the model. The thread execution time encapsulates the information about the interference in hardware resources, and about the benefits of data and instructions sharing between simultaneously running threads.

3.3.1.1 Phase 1: Application profiling

Application profiling is the first phase of TSBSched scheduler. In this phase, we measure the interference between concurrently running threads in the profiling set of thread assignments. The thread assignment method models two aspects of interferences between concurrently running threads: (1) Collision in shared hardware resources, and (2) Benefit of data and instruction sharing.

(1) Collision in shared hardware resources: Interference in shared processor resources between concurrently running threads depends on the hardware resources that the threads use. We did a detailed characterization of the resource sharing levels of the UltraSPARC T2 processor [154] as a representative of processors with several levels of resource sharing. The results of this analysis show that, during the workload selection, it is very important to consider the interference between threads in all levels of resource sharing – IntraPipe, IntraCore, and InterCore, in the case of UltraSPARC T2. On the other hand, once the workload is selected, the execution of threads running on core N is negligibly affected by the assignment of threads that do not run on core N (that run on remote cores). This fact significantly reduces the complexity of TSBSched scheduler. Instead of the analysis of all the threads running on the processor, the thread assignment method can reduce the scope of the analysis only to threads that execute on the same core.

The fact that we can reduce the scope of the analysis to the thread running on a single processor core, enables us to perform a brute force exploration, i.e. to execute all possible thread assignments inside the core. We execute each thread under study (the target thread) with all possible layouts of the workload inside the processor core. We measure the execution time of the target thread in each experiment, and store this data into the table. Since this data shows the slowdown that the target threads experience because of the collision with co-running threads, we refer to this table as the *Slowdown Table*.

We illustrate the experiments and the Slowdown Table with an example in which several IPFwd instances execute concurrently on the UltraSPARC T2 processor. The IPFwd is a low-layer network application comprised of three threads: Receiving (R), Processing (P), and Transmitting (T) (see Section 2.3). The UltraSPARC T2 core comprises two hardware pipelines, and each pipeline supports the concurrent execution of up to four threads.

In order to model the interference between threads that execute concurrently on the processor core, we run one *target application* R_{tg}-P_{tg}-T_{tg} with several instances of *stressing applications* R_{st}-P_{st}-T_{st}. Target and stressing applications perform the same processing of network packets. The only difference is that we monitor the performance of the target application, and do not monitor the stressing application instances. The purpose of the stressing applications is to cause the interference in shared processor resources that could affect the performance of the target application in a given thread assignment. In order to quantify the slowdown that R_{tg} thread experiences when it interferes with co-running threads, we execute thread assignments in which one *target thread* R_{tg} executes on the same processor core with all possible layouts of *stressing threads* R_{st}, P_{st}, and T_{st}; i.e. R_{tg} runs on the same core with threads: $[R_{st}]$; $[R_{st}R_{st}]$; $[R_{st}R_{st}R_{st}]$; $[P_{st}]$; $[P_{st}P_{st}]$; $[P_{st}P_{st}P_{st}]$; $[R_{st}P_{st}T_{st}]$; etc. The Slowdown Table for R_{tg} thread has as many entries as different layouts of *stressing threads* running on the same core with R_{tg}, and each entry contains the execution time of the *target thread* in a given thread assignment. In the experiments in which we profile R_{tg}, threads P_{tg} and T_{tg} are executed in isolation on remote cores. The experiments needed to characterize P_{tg} and T_{tg} threads are analogous to R_{tg} experiments.

(2) Benefit from data and instructions sharing: Threads running on the same core of the UltraSPARC T2 processor share the L1 instruction and data cache. Therefore, if several threads share data or instructions, they may benefit from co-scheduling on the same processor core.

In order to detect whether an application can experience performance improvement when the threads share L1 instruction and data cache, we execute a single application instance in all possible thread assignment and measure the execution time of each thread in each of the assignments. The results of these experiments are stored in the *Base Time Table*. The table has as many entries as there are different thread assignments of a single application, and each entry contains the execution time of each application thread in a given assignment.

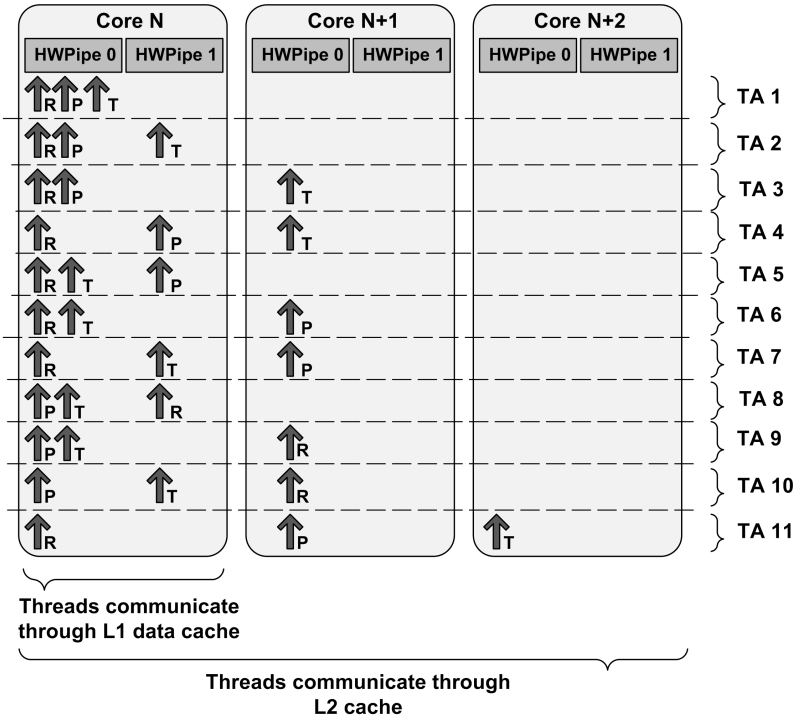


Figure 3.2: Experiments for the Base Time Table

We illustrate the experiments needed for the Base Time Table using the same example of 3-stage IPFwd application running on the UltraSPARC T2 processor. When consecutive IPFwd threads (Receiving and Processing, or Processing and Transmitting) execute on the same processor core, they communicate through L1 data cache. When the threads execute in different cores, the communication is through L2 cache, what causes additional L1 cache misses and the overhead in the application execution time.

In order to measure the impact of communication through L1 or L2 cache on application performance, we execute all possible thread assignments of a single IPFwd application instance (11 assignments in total), and measure the execution time of all IPFwd threads. The set of experiments is presented in Figure 3.2. For example, in the thread assignment 11 (*TA11*), R, P, and T threads execute in different cores. Therefore, in case that these threads share the data, any update of the values will be proceeded through globally-shared L2 cache. On the other hand, in *TA7*, R and T threads execute on the same core, thus any data update will be proceeded locally in L1 cache and it will not require invalidation of the cache lines in remote cores.

Application profiling phase *vs.* run-time: It is very important that the behavior of an application under study in the profiling phase is representative of its run-time behavior.

If the application can be configured by using different parameters, the parameters in the profiling phase have to correspond to the run-time parameters. If the application behavior depends significantly on the network traffic, the traffic used when the application is profiled should be representative of the run-time traffic. Also, in the profiling phase, the user could analyze the application behavior for different sets of parameters and different types of network traffic.

3.3.1.2 Phase 2: Performance prediction

Performance prediction is the second phase of the thread assignment method, see Figure 3.1. In this phase, based on the profiling data, the Base Time Table and Slowdown Table, TSBSched predicts the performance for *any* thread assignment composed of the applications that are analyzed in the application profiling phase. The output of performance prediction phase is the list of different thread assignments (thousands of them) with the predicted performance for each assignment. The performance prediction phase is comprised of three steps:

In *Step 1*, we analyze each application in the workload independently, as it executes in isolation. Hence, we disregard the interference between different applications that execute simultaneously on the processor, and model only the interaction between threads that belong to the same application. In this step, each application thread in the workload is associated with its *base_time*, the execution time that the thread would have had as if the application were executed in isolation.

In *Step 2* of the performance prediction, we model the effect of collision in hardware resources between different applications or different instances of the same application. In this step, we take into account the interference between all the threads in a given thread assignment.

Finally, in *Step 3*, based on the analysis in *Step 1* and *Step 2*, we compute the predicted performance of a given thread assignment.

We illustrate the application of TSBSched with an example thread assignment that is presented in Figure 3.3. The assignment is comprised of two IPFwd instances, R1-P1-T1 and R2-P2-T2, that execute on four processor cores, *Core N* to *Core N+3*. In this example, threads R1, R2, and P2 execute in the same hardware pipeline (HWPipe 0) in core N, while other threads execute in different processor cores.

Step 1: In *Step 1*, we analyze application instances R1-P1-T1 and R2-P2-T2 independently, as if each of them were executed in isolation. For example, when we analyze

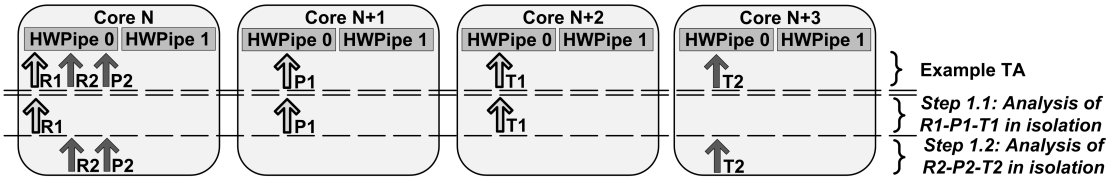


Figure 3.3: Performance prediction. Step 1: Analysis of each application instance in isolation.

R1-P1-T1 application we disregard R2, P2, and T2 threads from the original thread assignment, and vice versa (see Figure 3.3). In this step, each thread in the workload is associated with its *base_time*, the execution time that the thread would have had if a single application instance were executed alone on the processor. The *base_time* is read directly from the Base Time Table (see Section 3.3.1.1).

In *Step 1.1*, we analyze R1-P1-T1 application in the example thread assignment presented in Figure 3.3. We directly read the *base_time* of R1, P1, and T1 threads from the fields of the Base Time Table that corresponds to the thread assignment 11 (*TA11*) presented in Figure 3.2 (R, P, and T threads assigned to different cores). The *base_time* of R2-P2-T2 application is computed analogously. First we observe R2-P2-T2 application as if it were executed in isolation (Step 1.2 of Figure 3.3). Then we read the corresponding fields of the Base Time Table – the field that corresponds to *TA3* presented in Figure 3.2.

Step 2: In *Step 2*, we model the collision in hardware resources between threads that belong to different applications or different application instances. As we explained in Section 3.3.1.1, we focus on interference between threads that execute on the same processor core. All the information needed to quantify the slowdown because of the collision between threads that are co-scheduled on the same processor core is stored in the Slowdown Table. Figure 3.4 presents the part of the example thread assignment that we use to illustrate *Step 2* of the performance prediction. We analyze the interference between threads R1, R2, and P2 that execute on the same hardware pipeline of core N. Figure 3.4 present also the entries of the Slowdown Table that are used in the analysis.

In *Step 2.1* presented in Figure 3.4, we compute *slowdown(R1)*, the slowdown that the thread R1 experiences because of the interference with threads R2 and P2. The most important part of this analysis is to match the thread assignment under study with corresponding entries of the Slowdown Table. Since the thread R1 is the thread under study, it corresponds to the R_{tg} in the Slowdown Table. We want to detect the slowdown that the threads R2 and P2 cause to the R1 thread. Thus, the threads R2 and P2 correspond

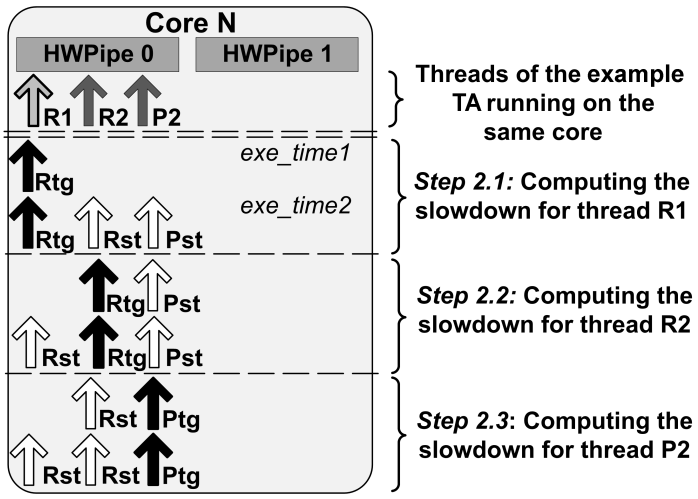


Figure 3.4: TSBSched performance prediction. Step 2: Modeling the collision in hardware resources.

to R_{st} and P_{st} threads in the Slowdown Table. First, from the Slowdown Table, we read the execution time of the target thread R_{tg} when it executes alone on the processor core – *exe_time1*. This entry corresponds to the thread assignment in which the thread R1 executes in isolation on the processor core. Later, we read the execution time of the thread R_{tg} when it executes on the same hardware pipeline with one R_{st} and one P_{st} thread – *exe_time2*. This corresponds to the thread R1 running with the threads R2 and P2 on the same hardware pipeline in a processor core. Finally, the slowdown that the thread R1 experiences because of the interference with the threads R2 and P2 is computed as: *slowdown(R1) = exe_time2 - exe_time1*. In all the experiments, the threads P_{tg} and T_{tg} are executed in isolation on remote cores.

Step 2.2 determines the slowdown that the thread R2 experiences because of interference with thread R1 that is running in the same core *slowdown(R2)*. In *Step 2.3*, we compute *slowdown(P2)*, the slowdown the thread P2 experiences because it collides in processor core resources with the thread R1. Since the threads R2 and P2 belong to the same application instance, the interference between them is modeled entirely in *Step 1* of the analysis. As we show in Figure 3.4, *Step 2.2* and *Step 2.3* are analogous to *Step 2.1*.

Threads P1, T1, and T2 of the example thread assignment execute alone on different processor cores (see Figure 3.3), thus they do not experience any slowdown because of collision in processor core resources.

Step 3: In *Step 3* of the performance prediction, we compute the predicted throughput

(performance) of the observed thread assignment. First, we compute the time each thread in the assignment requires to process or transfer a single packet (the execution time of a thread). The thread's execution time is the sum of its *base_time* computed in *Step 1* and the *slowdown* from *Step 2*. For example, execution time of thread R1 is computed as: *exe_time(R1) = base_time(R1) + slowdown(R1)*. Second, we compute the execution time of each application instance in the workload. As R, P, and T threads process a packet in a pipeline, the application execution time is determined by the thread that has the longest execution time. For example, the execution time of the application R1-P1-T1 is computed as: *exe_time(R1-P1-T1)* = MAX[*exe_time(R1), exe_time(P1), exe_time(T1)*].

From the application execution time, we compute the application throughput in processed Packets Per Second (PPS) as:

$$throughput(R1\text{-}P1\text{-}T1)[PacketsPerSecond] = \frac{CPUfreq[CyclesPerSecond]}{exe\_time(R1-P1-T1)[CyclesPerPacket]}$$

Finally, the performance of a given thread assignment is a sum of throughputs of all applications it is comprised of. In the Prediction phase, the model predicts the performance of thousands of different thread assignments. The output of TSBSched can be the thread assignment with the highest predicted performance, or, optionally, the prediction can be improved in the Selection phase.

3.3.1.3 Phase 3: Selection phase

Selection phase is the final phase of TSBSched, see Figure 3.1. Although TSBSched scheduler predicts the thread assignment performance with a high accuracy (as we show in Section 3.4), the thread assignment with the predicted highest performance could be wrongly-predicted. In order to avoid the performance loss in this case, in the Selection phase, the *actual* performance of several predicted best-performing thread assignments *is measured* on the target processor. The final outcome of TSBSched is the assignment with the highest *measured* (actual) performance. If the predicted best-performing thread assignment exhibits a low performance, the Selection phase will filter out this assignment, and the final outcome will be 2^{nd}, 3^{rd}, or N^{th} predicted best-performing assignment. In Section 3.4, we analyze also the impact of the Selection phase to the performance of TSBSched scheduler. The results show that the performance improvement of the Selection phase in which only five predicted best-performing thread assignments are executed on the real processor ranges up to 7.3%, which is significant.

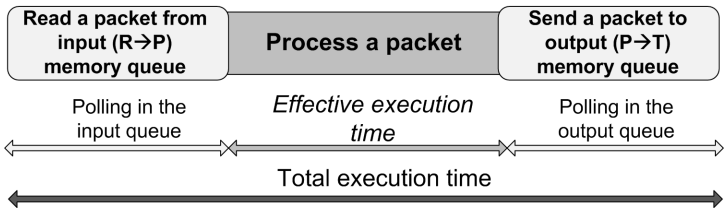


Figure 3.5: Execution time components of a Processing (P) thread

3.3.1.4 TSBSched limitations

TSBSched is a simple algorithm for thread assignment of multithreaded network applications running on processors with several levels of resource sharing. TSBSched has very high accuracy (as we show in Section 3.4), it is architecture independent, and does not need detailed knowledge about the application hardware requirements. However, there are two limitations that motive the enhancement of the application profiling and the prediction algorithm. These limitations are: (1) The programmer who profiles the application must have a profound knowledge about the application behavior; (2) As we will explain later, TSBSched requires changes in the application source code. If the source code is not available, TSBSched approach cannot be applied.

Figure 3.5 presents three different stages when a network packet is processed by IPFwd Processing (P) thread [11]: (1) First, the P thread reads the packet from the input (R→P) memory queue. If the input memory queue is empty, the P thread polls as long as there are no available packets in the queue. (2) Once the packet is read from the input queue, the P thread processes the packet. (3) After the processing, the packet is sent to the output (P→T) memory queue. As memory queues have limited capacity, the output queue may be full, so the P thread must poll until the following thread (T) processes a packet and frees an entry in the queue. Figure 3.5 also shows three components of total execution time of thread P: (1) The time that the P thread is polling in the input queue; (2) *Effective execution time*, the time needed for thread P to process a packet; and (3) The time that the P thread spends polling in the output queue.

The R and T threads may also poll if there are no available packets in input queues, or if output queues are full. The only difference is that the R thread reads packets from the network card and writes to the R→P queue, and the T thread reads packets from the P→T queue and sends them to the network card. The total execution time of the threads R and T has the same components as the P thread: (1) The time spent polling in the input

queue; (2) Effective execution time; and (3) The time spent polling in the output queue. The effective execution time of thread R is the time needed to read the packet from the network card and transfer the packet to the memory. The effective execution time of the T thread is the time needed to send the packet to the network over the network card.

When a thread polls, it does no useful work, but only waits for other threads to process packets. Therefore, TSBSched does not take into account the time a thread spends polling, but predict performance of thread assignments based on the *effective execution time* of each application thread. Measuring the effective execution time is difficult. The programmer who profiles the application has to distinguish between packet processing and polling, and insert test points for measuring effective execution time at the right place in the code. If the time a thread spends polling cannot be extracted from the total execution time, the TSBSched is unfeasible. As measuring effective execution time requires changes in the source code, TSBSched can be used only if the application source code is available.

In order to apply TSBSched to our study, we analyzed the source code and profiled all the benchmarks in the suite. Since, in our case, the complete source code of all the benchmarks was available, we were able to insert the test points for measuring of the benchmark effective execution time and we could apply TSBSched approach to our case study.

3.3.2 BlackBox scheduler

In order to overcome the limitations of the TSBSched, we designed *BlackBox scheduler*. In the application profiling phase, instead of measuring the effective execution time of each application thread, BlackBox scheduler observes the application's throughput. In the framework we use in our study, we determine the throughput of the application based on the execution time (total execution time, with polling) of the last thread in the R-P-T pipeline (the T thread). The throughput (in processed Packets Per Second) is the ratio between processor frequency (in Cycles Per Second) and the thread execution time (in Cycles Per Packet):

$$throughput[PacketsPerSecond] \quad = \quad \frac{CPUfreq[CyclesPerSecond]}{exe\_time(T\ thread)[CyclesPerPacket]}$$

There are two main enhancements of BlackBox scheduler with respect to the TSBSched: (1) BlackBox scheduler does not require profound knowledge about applications that are to be scheduled. In this approach, the application is seen as a black box, and the only in-

formation needed for optimal scheduling is the application throughput in different thread assignments. (2) Since BlackBox scheduler does not require inserting any test points in the application source code, it can be used when the source code is not available.

BlackBox scheduler comprises the same three phases as the TSBSched: (1) Application profiling, (2) Performance prediction, and (3) Selection phase (see Figure 3.1).

3.3.2.1 Phase 1: Application profiling

In the application profiling phase, we measure the interference between concurrently running threads. As in the TSBSched method, the application profiling phase has two outputs, *Base Time Table* and *Slowdown Table*.

The data stored in the Base Time Table models (quantifies) the interaction between threads that belong to the same application instance. As in the TSBSched method, the Base Time Table contains as many entries as there are different thread assignments for a single application running on a target processor. The only difference with respect to the TSBSched method is that, instead of effective execution time of each application thread, each entry of the table contains the application's throughput in a given thread assignment.

The Slowdown Table quantifies the collision in shared hardware resources between threads that belong to different application instances. The table contains the same entries as for the TSBSched thread assignment method. The difference is that, instead of the execution time of each thread, the table entries contain the application's throughput. For example, in order to quantify the collision in hardware resources of an R thread on the performance of R-P-T application, we executed one target thread R_{tg} in the same processor core with all layouts of *stressing threads* R_{st}, P_{st}, and T_{st}. The table has as many entries as different layouts of stressing threads running in the same core with R_{tg} and each entry of the table contains the throughput of the target application R_{tg}-P_{tg}-T_{tg} in a given thread assignment. In the experiments in which we profile the R_{tg} thread, threads P_{tg} and T_{tg} are executed in isolation on remote cores. The experiments for characterization of P_{tg} and T_{tg} threads are analogous to R_{tg} experiments.

It is very important that the behavior of an application under study in the profiling phase is representative of its run-time behavior. If the application can be configured by using different parameters, the parameters in the profiling phase have to correspond to the run-time parameters. If the application behavior depends significantly on the network traffic, the traffic used when the application is profiled should be representative of the

run-time traffic.

3.3.2.2 Phase 2: Performance prediction

In the Performance prediction phase, based on data stored in the Base Time Table and Slowdown Table, BlackBox scheduler predicts performance of any thread assignment. Performance prediction, as in TSBSched method, comprises three steps. In *Step 1*, we analyze each application instance independently, as it runs in isolation. In *Step 2*, we model the interference in hardware resources between simultaneously running threads that belong to different application instances. Finally, in *Step 3*, we use results from *Step 1* and *Step 2* to compute the predicted performance of the thread assignment under study.

We explain the performance prediction of BlackBox scheduler with the example thread assignment presented in Figure 3.3, the same thread assignment that we used to explain TSBSched method.

In *Step 1*, we analyze applications R1-P1-T1 and R2-P2-T2 independently. In this step, each application instance in the thread assignment is associated with its *base_throughput*, the throughput the application would have had if it were executed in isolation on the processor. The application's *base_throughput* is read directly from the Base Time Table. For the application R1-P1-T1, the *base_throughput(R1-P1-T1)* is read from the entry of the Base Time Table that corresponds to *TA 11* presented in Figure 3.2. The computation of the *base_throughput(R2-P2-T2)* for application R2-P2-T2 is analogous (see Figure 3.3).

In *Step 2*, we model the impact of collision in hardware resources on the application performance. As in TSBSched, we focus on interference between threads from different application instances that execute on the same processor core. Interference between threads that belong to the same application instance are modeled (quantified) already in *Step 1* of the analysis. Figure 3.6 presents the part of the example thread assignment that we use to explain *Step 2* of BlackBox scheduler performance prediction algorithm.

In *Step 2.1* presented in Figure 3.6, we compute *slowdown(R1-P1-T1 | R1)*, the slowdown that the application R1-P1-T1 experiences because thread R1 interferes with threads R2 and P2. First, from the Slowdown Table, we determine the throughput of the target application R_{tg}-P_{tg}-T_{tg} when threads R_{tg}, P_{tg}, and T_{tg} execute alone on processor cores – *throughput(R_{tg}-P_{tg}-T_{tg})*. Later, we read the throughput of the target application when thread R_{tg} executes on the same hardware pipeline with one R_{st} and one P_{st} thread – *throughput(R_{tg}-P_{tg}-T_{tg} | R_{tg} vs. $R_{st}P_{st}$)*. This corresponds to the thread R1 run-

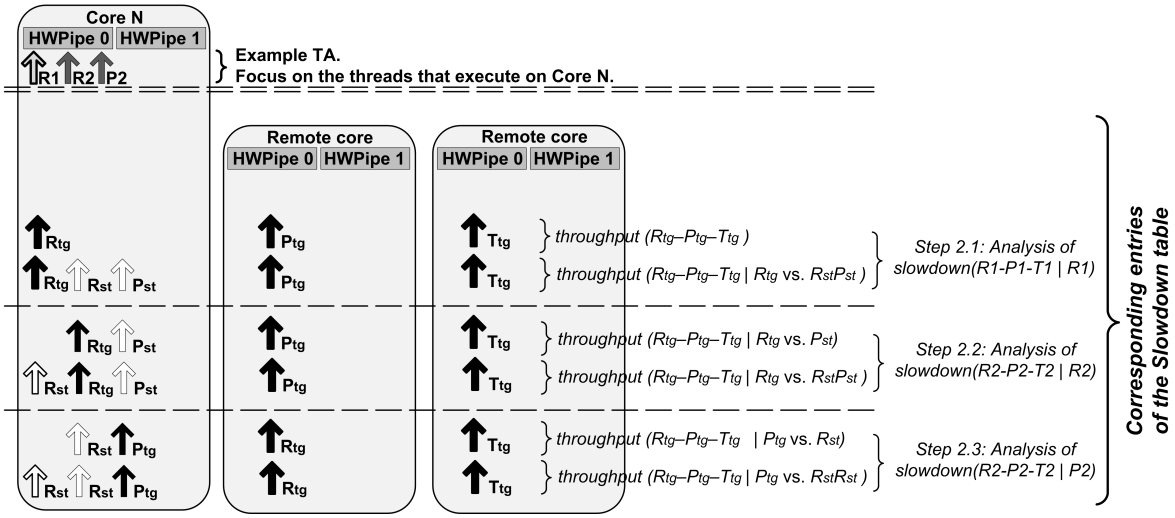


Figure 3.6: BlackBox scheduler performance prediction. Step 2: Modeling the collision in hardware resources.

ning with the threads R2 and P2 on the same hardware pipeline of processor core N. The *slowdown(R1-P1-T1 | R1)* is computed as:

$$\textit{slowdown(R1-P1-T1 | R1)} = \frac{throughput(R_{\text{tg}}{-}P_{\text{tg}}{-}T_{\text{tg}})}{throughput(R_{\text{tg}}{-}P_{\text{tg}}{-}T_{\text{tg}} \mid R_{\text{tg}} \ \ vs. \ \ R_{\text{st}}P_{\text{st}})}.$$

Since threads R1 and T1 execute in isolation on processor cores, *slowdown(R1-P1-T1 | P1)* and *slowdown(R1-P1-T1 | T1)* are equal to 1. Also, it is important to notice that *slowdown(R1-P1-T1 | R1)* may be equal to 1, even though R1 does not execute alone on the processor core. For example, assume that thread P1 is the bottleneck of the R1-P1-T1 application. Even if R1 thread experiences a slowdown because of collision with R2 and P2 running on the same core, P1 thread may still be the application bottleneck. In this case, *throughput(R_{tg}-P_{tg}-T_{tg})* will be equal to *throughput(R_{tg}-P_{tg}-T_{tg} | R_{tg} vs. $R_{st}P_{st}$)*, and *slowdown(R1-P1-T1 | R1)* will be equal to 1.

Step 2.2 determines the slowdown that the application R2-P2-T2 experiences because thread R2 interacts with thread R1 – *slowdown(R2-P2-T2 | R2)*. In *Step 2.3*, we compute *slowdown(R2-P2-T2 | P2)*, the slowdown that the application instance R2-P2-T2 experiences because thread P2 collides in shared CPU resources with thread R1. Interference between threads R2 and P2 is already modeled entirely in *Step 1* of the analysis. As we show in Figure 3.6, *Step 2.2* and *Step 2.3* are analogous to *Step 2.1*.

Thread T2 executes alone on a processor core, therefore *slowdown(R2-P2-T2 | T2)* equals 1.

In *Step 3*, we compute the predicted throughput of the thread assignment. First, for each application instance Ri-Pi-Ti we read the *base_throughput(Ri-Pi-Ti)*, as we describe in *Step 1*. Later, we compute *slowdown(Ri-Pi-Ti | Ri)*, *slowdown(Ri-Pi-Ti | Pi)*, and *slowdown(Ri-Pi-Ti | Ti)*, the slowdown that the application Ri-Pi-Ti experiences because threads Ri, Pi, and Ti interfere in hardware resources with other application instances, as we explain in *Step 2*. Since Ri, Pi, and Ti threads process packets in a software pipeline, the application performance are determined by the slowest thread in a given thread assignment - the bottleneck thread. Therefore, the slowdown that the application Ri-Pi-Ti experiences because of the collision in hardware resources corresponds to the highest slowdown computed in *Step 2*:

slowdown(Ri-Pi-Ti)=MAX[*slowdown(Ri-Pi-Ti | Ri), slowdown(Ri-Pi-Ti | Pi), slowdown(Ri-Pi-Ti | Ti)*].

The throughput of each application is determined with its *base_throughput* and *slowdown*:

$$\textit{throughput(Ri-Pi-Ti)} = \frac{base\_throughput(Ri-Pi-Ti)}{slowdown(Ri-Pi-Ti)}.$$

Finally, the performance of a given thread assignment is a sum of throughputs (measured in processed Packets Per Second) of all application instances it comprises.

After the prediction phase, each thread assignment from the input set is related with its predicted performance. The output of the BlackBox scheduler may be the input thread assignment with the highest predicted performance, or, optionally, the prediction of the model can be improved in the Selection phase.

3.3.2.3 Phase 3: Selection phase

As in the TSBSched method, in the Selection phase, we execute the N best-predicted thread assignments on the target processor and monitor the actual performance of each assignment. The thread assignment with the highest measured performance is the final outcome of the thread assignment method.

3.3.3 Scalability of the thread assignment methods

The data used to predict the performance of thread assignments is stored in the Base Time Table and the Slowdown Table. In order to collect data for the Base Time Table, we have to execute all thread assignments of a single application on the target processor. The Slowdown Table requires running all possible layouts of application threads on a single processor core.

Since state-of-the-art networking applications comprise few threads [11][155], the

Table 3.1: Scalability of the proposed thread assignment method

Application threads	Application instances	Total threads	Number of input experiements			Experimentation time	
			Base Time Table	Slowdown Table	Total	1 server	4 servers
4	8	32	49	9,800	9,849	5.5 hours	1.3 hours
	16	64					
6	5	30	1,526	105,840	107,366	2.5 days	15 hours
	10	60					
8	4	32	74,376	653,400	727,776	17 days	4.2 days
	8	64					
32	1	32	1.4×10^{31}	4.7×10^{8}	1.4×10^{31}	9×10^{23} years	2.3×10^{23} years
	2	64					

execution of experiments needed to fill the Base Time Table and the Slowdown Table is feasible, and the proposed thread assignment methods can be applied. Most of the low-layer network applications are composed of few threads because the packet processing is short, so splitting it into many threads introduces communication overheads that overcome the benefits of multithreading.

It is important to notice that the application profiling is one-time effort, and that the process can be fully automated, thus the execution of the experiments and the data processing require no interaction with the programmer. In Table 3.1, we present the number of profiling experiments needed to characterize workloads comprised of different number of threads running on the UltraSPARC T2 processor. We use this analysis to understand whether TSBSched and BlackBox scheduler can be used if the number of application threads increases. The first column of Table 3.1 lists the number of threads that comprise a single application instance. The second and the third column show the number of application instances and the total number of the threads in the workload (Total threads = Application threads $\times$ Application instances). The following columns show the number of input experiments needed for the Base Time Table and the Slowdown Table, respectively. Finally, the last two columns of the table show the time required to execute all the profiling experiments. The experimentation time is calculated using the assumption that a single experiment can be executed in two seconds, which is correct in our experimental environment. We present results when the experiments can be executed on a single server, and when four servers can be used to simultaneously execute the experiments.

We reach several conclusions from the results presented in Table 3.1. For four, six, and eight application threads, the time needed to execute the profiling experiments on a single server is 5.5 hours, 2.5 days, and 17 days, respectively. However, as the experiments

needed for application profiling are independent, they can be executed simultaneously on *N* servers which will reduce the experimentation time by *N* times. For example, when four servers are used, the profiling experiments for applications that are comprised of four, six, and eight threads can be executed in 1.3 hours, 15 hours, and 4.2 days, respectively, which is feasible in most of the industrial environments. Also, it is important to notice that the number of the profiling experiments does not increase with the total number of threads in the workload, but with the number of threads that compose a single application instance. The proposed thread assignment methods can be used to determine good thread assignments for fully-utilized processor (60 to 64 simultaneously running threads) as long as the number of threads that compose a single application instance is not high.

For the applications that are comprised of a large number of threads, running all profiling experiments becomes infeasible. For example, for application that comprise 32 threads, running all the profiling experiments would require 9×10^{23} years. If the workload is composed of different multithreaded applications, TSBSched and BlackBox scheduler also have to consider interference between thread that belong to different applications. In this case, the profiling experiments needed to fill the Base Time Table would not change. For each application, the Base Time Table would be constructed as the application is to be executed in isolation (see Section 3.3.1.1). On the other hand, including a new application in the workload would require new experiments that would extend the existing Slowdown Table. The experiments needed to fill the Slowdown Table require the execution of all possible layouts of the workload inside the processor core.

Table 3.1 shows the number of input experiments for UltraSPARC T2 processors that comprise eight cores and eight hardware contexts per core. The number of profiling experiments is significantly lower for processors with lower number of cores and hardware contexts per processors core.

3.4 Evaluation

In this section, we present the evaluation of TSBSched and BlackBox scheduler for the set of benchmarks presented in Section 2.3. In order to evaluate the proposed thread assignment methods, we execute few application instances at once in different thread assignments, and measure the performance of each assignment. Then, we compare the performance of best-predicted thread assignment with the performance of actual best one observed in the evaluation sample. We also show how the Selection phase additionally improves the performance delivered by the proposed scheduling methods. Finally, we

present the improvement that our methods obtains over a naive scheduling and a resource-oblivious scheduling algorithm used in current OSs.

As a main metric, we use the number of Packets Per Second (PPS) processed by different applications. This metric is inversely proportional to the packet processing time, and has the same properties as the execution time for non-packet oriented applications.

3.4.1 Exploration space

We evaluated TSBSched and BlackBox scheduler for 2, 3, 4, 6, and 8 instances of benchmarks presented in Section 2.3, i.e. for 6, 9, 12, 18, and 24 concurrently running software threads. We could not execute more than 24 threads (eight benchmark instances) simultaneously because of the limitation of 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 may bind at most one receiving thread (one benchmark instance) to each DMA channel.

The number of different thread assignments composed of six software threads is around 1500. We generated all thread assignments, executed them on the real hardware and measured the performance for each of them. Then, we compared the actual best assignment with the ones predicted by the TSBSched and BlackBox scheduler.

For nine threads running on the UltraSPARC T2 processor, the number of different assignments exceeds half million, so it is unfeasible to run all of them on the processor because of the execution time constrains. As an alternative to the brute-force exploration we used a statistical sampling. Even the sampling does not ensure that the best-performing thread assignment in the sample is also the best assignment in the whole population (the set of all possible assignments), this is statistically correct method to determine set of thread assignments that we used to evaluate TSBSched and BlackBox scheduler.

We used systematic sampling [44], a sampling method where the target population, all thread assignments in our case, is ordered by some property and then the samples are generated by selecting elements at regular intervals through the ordered list. The first element is randomly selected from the first k elements and then every k^{th} element is selected until the end of the list. The sampling step k is computed as $k = \frac{\text{population size}}{\text{sample size}}$. Because of execution time constraints, we chose a sample size of 700 elements which gives the sampling step $k = 750$.

We generated all possible thread assignments and used TSBSched to predict their performance. Then, we sorted them by predicted performance and did the sampling. Sys-

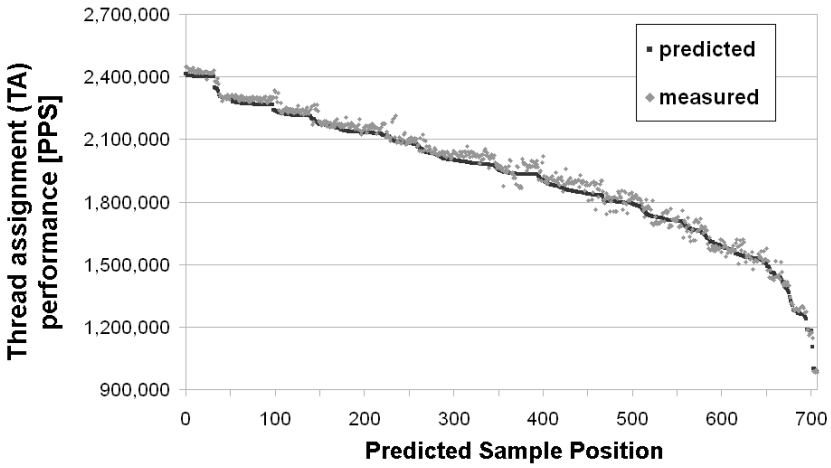


Figure 3.7: Comparison of predicted and measured performance for a chosen sample (nine *IPFwd-intadd* threads)

tematic sampling is effective if the population is sorted by a property that is related to the measured value. Ideally, the population should be sorted by the value we want to measure. In our case, we sorted the population by performance predicted with TSBSched. In order to prove the validity of our sample, we had to show that the predicted performance are closely related to the actual ones. Figure 3.7 shows predicted and measured performance for a sample of nine simultaneously running *IPFwd-intadd* threads. The X-axis shows the order of the thread assignment in the sample, while Y-axis presents the performance of the assignment. Since our benchmark suite is composed of network applications, we measure the performance in processed Packets Per Second (PPS). The results presented in the figure show very small difference between predicted and measured performance. We compared the predicted and measured performance for all benchmarks in the suite, and detected that the prediction closely followed the measured results in all the cases. Therefore, we concluded that the obtained samples are valid and that systematic sampling is effective.

The number of different thread assignments for 12, 18, and 24 software threads exceeds hundreds of millions. Because of time constrains, it is not feasible to generate nor predict all of them. Therefore, out of all possible thread assignments, we randomly sampled 1000 assignments [44], and verified our thread assignment methods for a given sample. The method we used to generate random thread assignments is described next.

For example, assume that a workload of T threads would be assigned on a processor comprising V hardware contexts, where $T \leq V$. We enumerated the hardware contexts of

the processor with integers from 1 to V and for each thread in the workload we randomly selected an integer from this interval using the uniform distribution. The number assigned to each running thread represented the hardware context to which the thread was mapped to in a given thread assignment. After mapping all the threads, we checked whether the generated thread assignment was valid. An assignment is not valid if two or more threads are mapped to the same hardware context. If this is the case, we simply discarded the invalid assignment and repeated the whole process until the sample contained the required number of thread assignments.

The sampling method generated thousands of random assignments in a matter of seconds. However, we are aware that the presented approach may not be effective for generation of random assignments comprised of a large number of software threads. As the number of software threads approaches the number of hardware contexts, the probability that more than one thread are mapped to the same hardware context (the assignment in not valid) increases. In order to avoid the case when most of the generated assignments are invalid and discarded, the one could think about more effective approaches to generate a random sample of thread assignments.

Since the number of input thread assignments may be very large, we pay special attention to make the performance prediction phase of TSBSched and BlackBox scheduler as fast as possible. Both approaches predict performance for up to 5000 thread assignments per second [1]. This number is constant for different number of assignments under study, and it is inversely proportional to the number of threads in the workload.

3.4.2 Six software threads

The results for six software threads (two application instances) are presented in Figure 3.8. Different benchmarks are listed along X-axis of the figure, while Y-axis shows the slowdown with respect to the best actual thread assignment (TA). We present few groups of bars. *TSBSched* and *BlackBox* bars shows the performance difference between the actual best thread assignment and the ones predicted by TSBSched and BlackBox scheduler, respectively. In these experiments, the Selection phase is not included. Bars *balanced-average* and *balanced-worst* present the average and the worst slowdown of balanced thread assignment with respect to the actual best one. Balanced assignments have threads equally distributed between processor hardware domains. Current state-of-the-art schedul-

[1] The models were executed on Intel Dual Core processor running at the frequency of 1.83GHz and 1GB of memory.

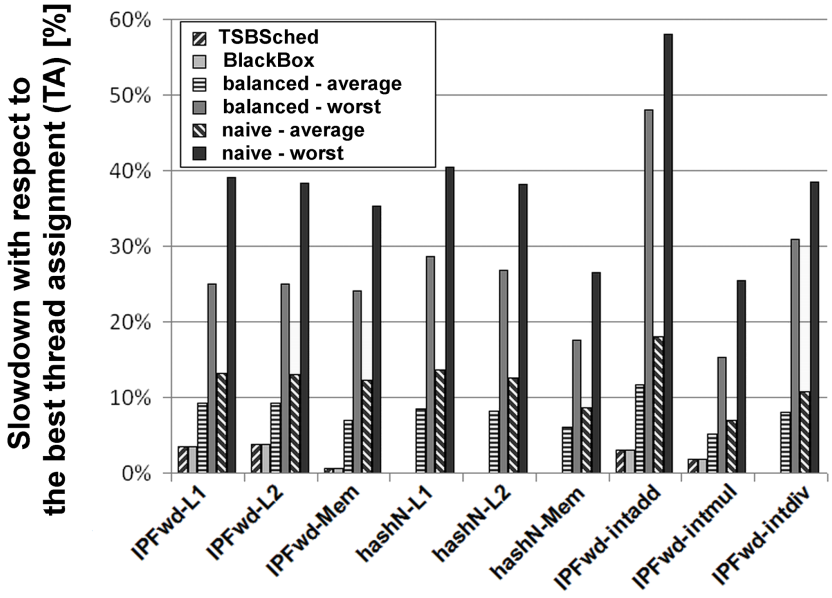


Figure 3.8: Slowdown with respect to the best thread assignment (six software threads)

ing algorithms used in Linux and Solaris [28] [134] try to equally distribute co-running threads between different hardware domains (for UltraSPARC T2 processor, between processor cores and hardware pipes). The aim of this approach is to distribute co-running threads to equally stress hardware resources. Since current Linux and Solaris load balancing algorithms disregard inter-thread dependencies and different resource requirements of each thread, the performance of balanced thread assignments may not be the optimal. Finally, bars *naive-average* and *naive-worst* present the average and the worst slowdown of naive scheduling with respect to the actual best thread assignment.

For six concurrently running software threads running and the set of benchmarks we used in the study, TSBSched and BlackBox scheduler have precisely the same accuracy. Both approaches determine thread assignments that have performance very close to the optimal ones. TSBSched and BlackBox scheduler predict the absolute best thread assignment for four out of nine applications: *hashN-L1*, *hashN-L2*, *hashN-Mem*, and *IPFwd-intdiv*. The performance loss of the best predicted assignment with respect to the actual best one is always below 4%.

The improvement of our assignment methods with respect to other scheduling techniques is significant. The performance gain with respect to Linux-like scheduler is between 3% and 9% in average, and it ranges up to 45% in the worst case (*IPFwd-intadd* benchmark). The performance gain with respect to a naive process scheduling ranges

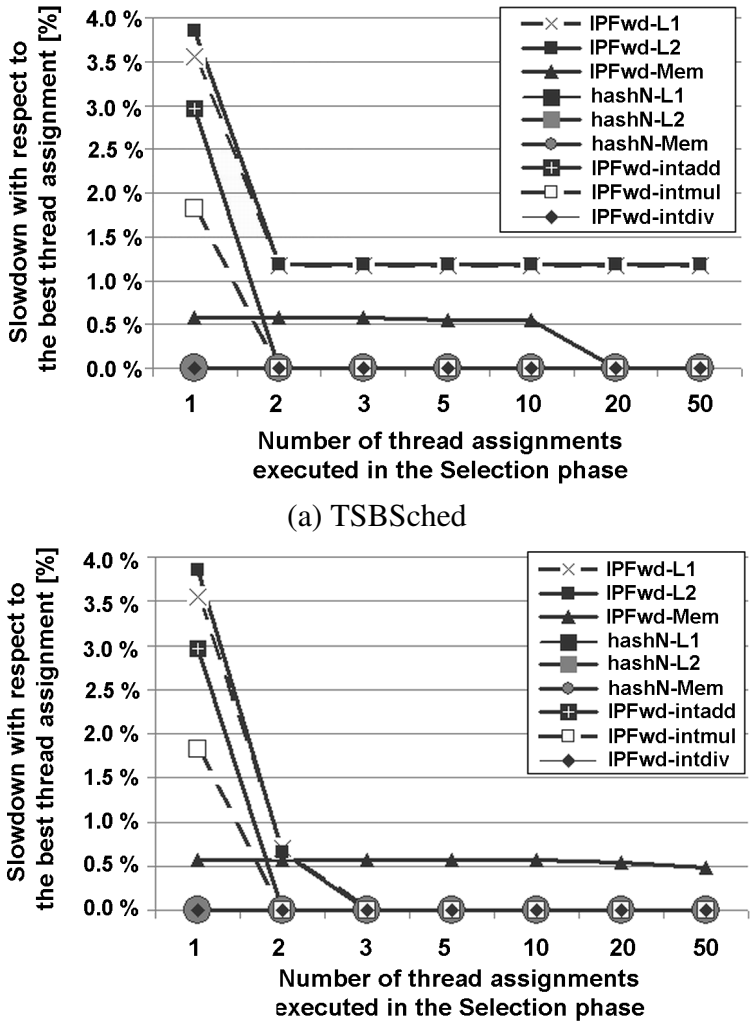
between 5% and 15% in average; in the worst case it is between 24% and 55%.

Figure 3.8 presents results for the TSBSched and BlackBox scheduler when the Selection phase is not included, but the output of the scheduler is the best predicted thread assignment. Figure 3.9 shows performance improvement when the models includes the Selection phase. The X-axis of the charts shows the number of thread assignments executed on in the Selection phase, while Y-axis shows the slowdown with respect to the best actual assignment. The results presented on Figure 3.9 show that running only two or three thread assignments in the Selection phase can additionally improve the prediction of presented methods. For example, for TSBSched (see Figure 3.9(a)), the Selection phase with only two best predicted thread assignments improves the prediction for four out of nine benchmarks. In four out of remaining five benchmarks, the best predicted assignment is also the best actual one, so the performance cannot be improved because it is already the highest possible. Performance improvement of the Selection phase for TSBSched approach ranges from 1.8% (*IPFwd-intmul*) to 3% (*IPFwd-intadd*). For BlackBox scheduler (see Figure 3.9(b)), the improvement of the Selection phase ranges up to 3.9% (*IPFwd-L2* benchmark). When BlackBox scheduler includes the Selection phase, running only three best predicted thread assignments is enough to capture the best actual assignment for eight out of nine benchmarks.

3.4.3 Nine software threads

The results for three application instances are presented in Figure 3.10. Different benchmarks are listed along X-axis of the figure, while Y-axis shows slowdown with respect to the best observed thread assignment. We present few groups of bars. Bars *TSBSched-TOP1* and *BlackBox-TOP1* show the performance difference between best observed thread assignment and the one predicted by TSBSched and BlackBox scheduler without the Selection phase. These two bars show the performance difference between the best predicted and the best observed thread assignment. Bars *TSBSched-TOP5* and *BlackBox-TOP5* show the performance difference between the best observed thread assignment and the one predicted by TSBSched and BlackBox scheduler, but this time with five best-predicted assignments executed in the Selection phase. The rest of the bars is the same as in Figure 3.8: bars *balanced-average* and *balanced-worst* present the average and the worst slowdown incurred by Linux-like scheduler; the values of *naive-average* and *naive-worst* bars present the average and the worst slowdown incurred by naive scheduling.

First, we analyze the results of TSBSched. When the Selection phase is not included



(a) TSBSched

(b) BlackBox scheduler

Figure 3.9: The improvement of the Selection phase (six software threads)

(bar *TSBSched-TOP1*), TSBSched method captures the actual best thread assignment for three out of nine benchmarks (*hashN-L1*, *IPFwd-intmul*, and *IPFwd-intdiv*). The highest performance loss of TSBSched is only 1.9% (*IPFwd-L2*). The Selection phase additionally improves the prediction for *IPFwd-Mem*, *hashN-Mem*, and *IPFwd-intadd* benchmarks, so the model predicts the actual best assignment for six out of nine benchmarks (see bar *TSBSched-TOP5*).

The results for BlackBox scheduler are even better. When the Selection phase is not included (bar *BlackBox-TOP1*), the best predicted assignment is also the observed best one in four cases (*IPFwd-Mem*, *hashN-L1*, *IPFwd-intmul*, and *IPFwd-intdiv*). The highest performance of loss is only 0.4% (*IPFwd-L2* benchmark). Again, the Selection

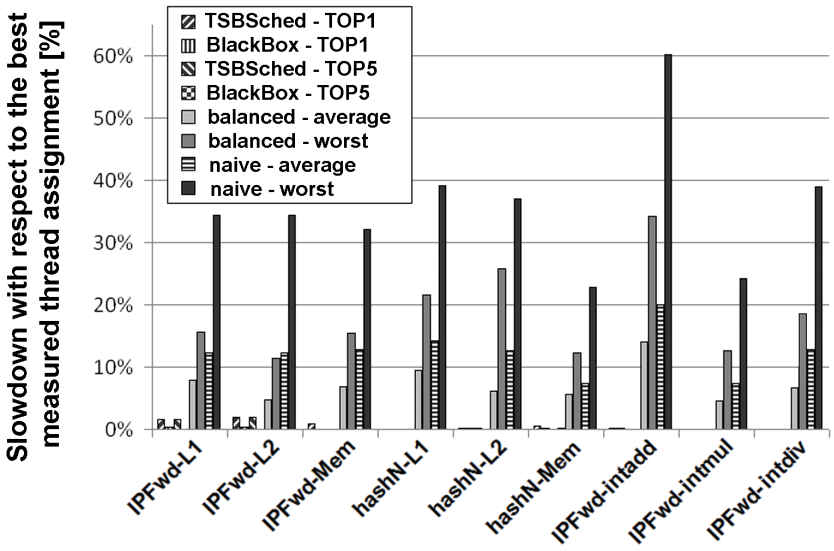


Figure 3.10: Slowdown for nine software threads

phase additionally improves the prediction of the model: BlackBox scheduler predicts the actual best thread assignment in the evaluation sample for eight out of nine benchmarks, all except *hashN-Mem* (see bar *BlackBox-TOP5*). The only time the model does not determine the actual observed assignment, the performance loss is only 0.1%.

TSBSched and BlackBox scheduler significantly improve the performance of the state-of-the-art OS schedulers and a naive scheduler. The performance gain with respect to the Linux-like scheduler is between 5% and 14% in average, and up to 34% in the worst case (*IPFwd-intadd*). The performance gain with respect to naive process scheduling is even higher; it ranges from 7% to 20% in average, and from 23% to 60% in the worst case.

3.4.4 12, 18, and 24 software threads

The results for 12, 18, and 24 software threads are presented in Figure 3.11. Different benchmarks are listed along X-axis of figures, and Y-axis shows the slowdown with respect to the best observed thread assignment in the evaluation sample.

We present few groups of bars. Bars *TSBSched-TOP1* and *BlackBox-TOP1* show the performance difference between actual observed thread assignment and the one predicted by TSBSched and BlackBox scheduler without the Selection phase. Bars *TSBSched-TOP5* and *BlackBox-TOP5* show the same, but this time with the best five predicted thread assignments executed in the Selection phase. The values of *naive-average* and *naive-worst* bars present the average and the worst slowdown incurred by naive scheduling. For 12,

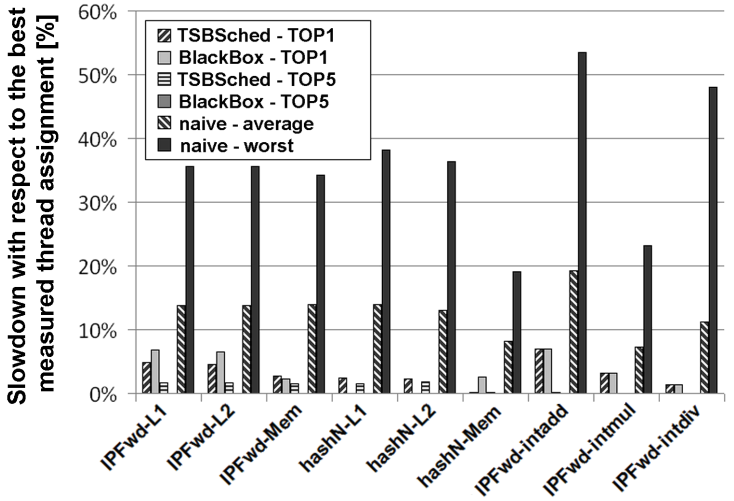
18, and 24 software threads, we evaluate the TSBSched and BlackBox scheduler for a random sample of 1000 thread assignments (see Section 3.4.1). Out of 1000 random assignments, in experiments for 12 software threads only 15 assignments were balanced, that was insufficient to report statistically significant results. Experiments for 18 and 24 threads did not include a single balanced assignment. Therefore, for these sets of experiments, we did not analyze the performance of balanced thread assignments (Linux-like scheduler).

Results presented in Figure 3.11(a), (b), and (c) have the same trend as results for six and nine software threads. Both TSBSched and BlackBox scheduler, show very high accuracy and introduce significant performance improvement with respect to naive scheduling. For 12 threads (see Figure 3.11(a)), the improvement of TSBSched and BlackBox scheduler with respect to naive scheduler is between 7% and 19% in average, and it reaches 54% in the worst case. For 18 threads (see Figure 3.11(b)), the performance gain of the proposed thread assignment algorithms ranges from 8% to 23% in average, and up to 50% in the worst case. In experiments with 24 simultaneously running threads, the improvement with respect to naive scheduling is between 6% to 19% in average, and up to 45% in the worst case (see Figure 3.11(c)).

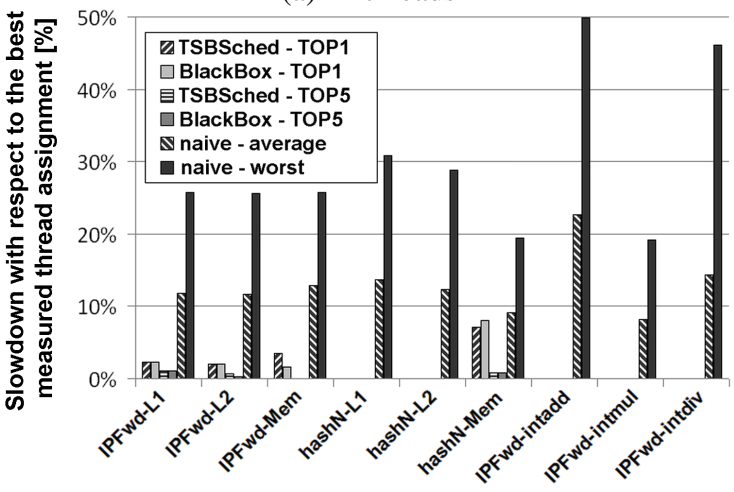
The results also show the importance of the Selection phase. First, we analyze the results for 12 threads and TSBSched, see Figure 3.11(a). Bar *TSBSched-TOP1* shows that, even most of the time the best predicted thread assignment is very close to the actual best one, for benchmarks *IPFwd-L1*, *IPFwd-L2*, and *IPFwd-intadd*, we measure the slowdown of 4.8%, 4.6%, and 7.0%, respectively. When the Selection phase is included (see bar *TSBSched-TOP5*), the slowdown for *IPFwd-L1*, *IPFwd-L2*, and *IPFwd-intadd* decreases significantly to only 1.7%, 1.7%, and 0.1%. The results for BlackBox scheduler are similar. BlackBox scheduler without the Selection phase incurs the slowdown of 6.8%, 6.5%, and 7.0% for *IPFwd-L1*, *IPFwd-L2*, and *IPFwd-intadd* benchmarks, respectively (see bar *BlackBox-TOP1*). BlackBox scheduler with the Selection phase, detects the best *actual* thread assignment in the evaluation sample for all nine benchmarks in the suite (see bar *BlackBox-TOP5*).

We detect similar behavior for 18 software threads, see Figure 3.11(b). For example, for *hashN-Mem* benchmark, TSBSched and BlackBox scheduler without the Selection phase introduce the slowdown of 7.2% and 8.1%. When the Selection phase is included, the slowdown is only 0.8% for both methods.

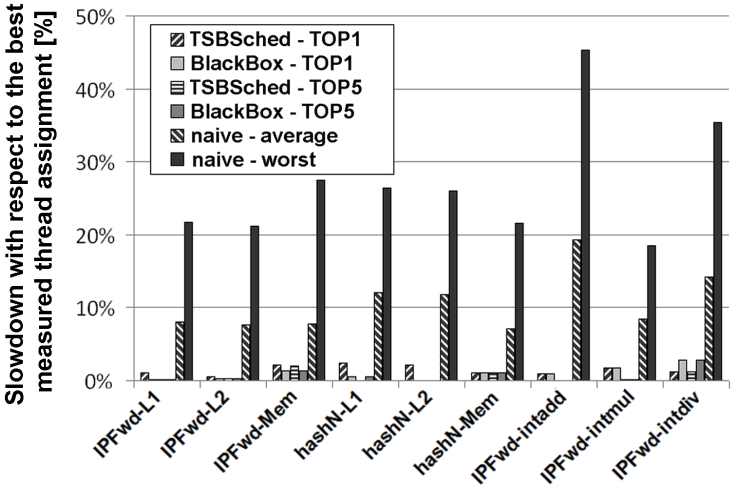
The results for 24 software threads are presented in Figure 3.11(c). Although TSBSched



(a) 12 threads



(b) 18 threads



(c) 24 threads

Figure 3.11: Slowdown for 12, 18, and 24 threads

and BlackBox scheduler without the Selection phase are very precise and the highest slowdown they introduce is only 2.8%, the Selection phase again improves the prediction. In TSBSched method, the Selection phase improves the prediction for six out of nine benchmarks, and increases the performance up to 2.5%. The Selection phase in BlackBox scheduler improves the prediction for *IPFwd-intadd* and *IPFwd-intmul* benchmarks for 1% and 1.6%, respectively.

3.5 Related Work

Workload Selection: Studies that address the workload selection problem propose models that predict the impact of interferences between co-running threads to system performance. Snavely et al. [144][143] present the SOS scheduler, the approach that uses hardware performance counters to find schedules that exhibit good performance. Eyerman and Eeckhout [62] propose probabilistic job symbiosis model that enhances the SOS scheduler. Based on the cycle accounting architecture [61][107][106], the model estimates the single-threaded progress for each job in a multithreaded workload.

Other approaches [40, 64, 91] propose techniques to co-schedule threads that exhibit a good symbiosis in shared caches solving the problems of cache contention. Kihm et al. [91] explore the design of a hardware monitoring system to produce the least interference in shared cache memory in every scheduling interval. Chandra et al. [40] present several performance models to predict the impact of cache sharing on co-scheduled threads. Fedorova et al. [64] implement a cache-fair scheduling algorithm in the OS. The algorithm reduces application performance variabilities because of interference in cache with co-runners. Cache-fair scheduling algorithm ensures that the application always runs as fast as it would under fair cache allocation, regardless of how the cache is actually allocated.

Several studies [86][90][108] demonstrate the drawbacks of (pseudo) *least-recently-used* (LRU) replacement policy in shared last level caches of multithreaded architectures. The studies also propose alternative replacement policies that consider workloads composed of (numerous) software threads with diverse cache requirements. Other studies [127][165] try to optimize the utilization of shared caches by adaptive spatial partitioning of these resources between simultaneously-running threads. Zhan et al. [166][167] propose cache management techniques that consider both issues – spacial partitioning and replacement policy. First, these techniques spatially partition cache resources between simultaneously-running threads to maximize the cache utilization. Then, cache replace-

ment policy for each running thread is adjusted to optimally utilize the cache resources that are allocated for that thread.

Doucette and Fedorova classify applications regarding their usage of shared processor resources [54]. The authors co-schedule applications with *base vectors* (micro-benchmarks that specifically stress a single CPU resource), and measure the slowdown application and base vector experience. This data is used to predict the performance of any set of applications concurrently running on the processor.

Kwok and Asmad [99][100] 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 interferences in shared processor resources, which is the focus of our study.

Several studies analyze the hardware support for control of inter-thread interferences in high-performance computing [27][38][151] and real-time systems [36][37][39]. These studies use techniques such as dynamic resource partitioning and thread priorities to improve the performance of a given workload running on the target architecture.

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 [16][59]. Acosta et al. [16] propose a thread 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. [59] also focus on CMP+SMT processors and propose an algorithm that uses hardware performance counters to profile thread behavior and assign compatible threads on the same SMT core. Other studies analyze thread scheduling for platforms comprised of several multithreaded processors [109][147]. McGregor et al. [109] introduce new scheduling policies that use run-time information from hardware performance counters to identify the best mix of threads to run across processors and within each processor. The main limitation of these studies is that the application profiling and thread scheduling is based on the measurements from numerous hardware performance counters. Since the set of available performance counters is specific for a given processor, it is very difficult to modify the proposed approaches to any new processor under study. This is the main reason why we do not use any of these approaches as a baseline in the evaluation of the TSBSched and BlackBox scheduler.

Kumar et al. [98] and Shelepov et al. [139] 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. Our study explores interferences between threads that are distributed between the *homogeneous* hardware domains (processor cores) of a processor.

Other studies propose solutions for optimal assignment of network workloads in network processors. Kokku et al. [96] propose an algorithm that assigns network processing tasks to processor cores with the goal of reducing the power consumption. Wolf et al. [164] 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 thread assignment solutions based on the application profiling and traffic analysis.

We present TSBSched and BlackBox scheduler, thread assignment methods for network applications running on multithreaded processors with several level of resource sharing. The proposed methods predict the performance of different thread assignments of a given workload based on minimum information about the target processor architecture, and without any data about the hardware requirements of the applications under study. To the best of our knowledge, TSBSched and BlackBox scheduler are the first methods that address thread assignment problem for multithraded applications.

3.6 Summary

Integration of different TLP models in the same processor increases resource utilization and improves the overall system performance. This motivates most processor vendors to combine different TLP paradigms in their latest designs.

On the other hand, this introduces complexities in the process scheduling. In additional to workload selection, process scheduling for processors with different levels of resource sharing requires the assignment of threads that compose the workload to the hardware contexts (virtual CPUs) of the processor. Thread assignment determines which hardware resources are shared between different threads, what directly affects the inter-threads interference in shared processor resources.

In this chapter, we demonstrated the importance of thread assignment and possible performance loss of non-optimal thread distribution. We presented TSBSched and BlackBox scheduler, simple methods for thread assignment of multithreaded network applications running on processors with several levels of resource sharing. We evalu-

ated TSBSched and BlackBox scheduler for a set of network applications running on the UltraSPARC T2 processor. The presented results show very high accuracy of both proposed thread assignment methods. In most of the experiments, both methods find the best observed thread assignment in the evaluation sample. The performance improvement of TSBSched and BlackBox scheduler ranges up to 48% with respect to the Linux-like scheduler, and up to 60% with respect to a naive thread assignment.

Chapter 4

A statistical approach to thread assignment problem

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

Due to the fact that the performance of the best possible thread assignment is unknown, the room for improvement of current thread-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 thread 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 thread assignment algorithms.

In this chapter, we present a method based on Extreme Value Theory that allows the prediction of the performance of the optimal thread assignment in multithreaded processors. We further show that executing a sample of several hundred or several thousand *random* thread 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.

4.1 Introduction

In multithreaded processors with large numbers of cores and several levels of resource sharing that, finding a good thread assignment becomes an intractable problem. As the number of possible thread assignments is vast (*e.g.* 10^{50}) [50, 59, 87, 131], it is unfeasible to do an exhaustive search in order to find the thread assignment with the highest performance. Also, the analytical analysis of optimal thread assignment is an NP-complete problem [67].

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

In this chapter, we present a method based on Extreme Value Theory (EVT) that allows the prediction of the performance of the optimal thread 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* thread 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 multithreaded architectures. Unlike other thread assignment proposals that use performance predictors to find the best thread 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:

- We show that running a sample of several hundred or several thousand random thread assignments will most probably capture an assignment within 1% or 2% of the best performing ones.

- We present a statistical method that, based on the measured performance of a sample of random assignments, estimates the performance of the best thread assignment *i.e.* the optimal system performance for a given workload.

- We present an iterative algorithm that, based on the previous analysis, samples random thread 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 threads-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 thread 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 thread 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 thread assignments with performance very close to the optimal one.

The rest of the chapter is organized as follows. Section 4.2 quantifies the number of possible different assignments in modern multithreaded processors and demonstrates the importance of knowing the optimal system performance. Section 4.3 presents a statistical analysis that we use to estimate the optimal system performance based on a small sample of random thread assignments. In Section 4.4, 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 4.5 presents the related work, while Section 4.6 lists the summary of the study.

4.2 Motivation and background

Importance of knowing the optimal system performance: Numerous studies present the algorithms for thread assignments on multithreaded processors (see Section 3.5). Given that, in general, running all possible thread assignments is unfeasible, several authors [16, 59, 131] verify their proposals with respect to 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 *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 thread assignment.

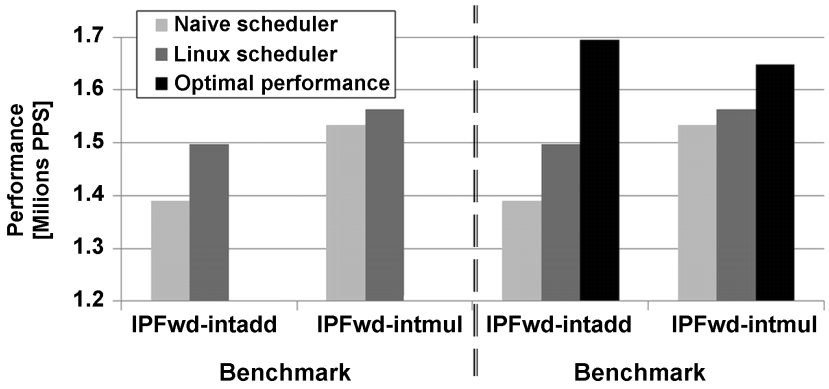


Figure 4.1: Comparison of a naive, Linux-like, and optimal thread assignment

We use an example to show how the evaluation of thread assignments techniques can be misleading if the optimal system performance is unknown. Figure 4.1 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 [131]. Figure 4.1 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 thread 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* thread assignment. In the right part, we also include the bar that corresponds to the optimal system performance.

Based on the results presented in Figure 4.1, 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 improvement 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 thread 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 thread assignment techniques, but it also shows the performance difference between the proposed thread assignments and the optimal one. This performance difference determines the upper limit for improvement of a given thread assignment algorithm, which is the most important piece of information for the system designer when deciding whether the algorithm should be enhanced.

4.3 A statistical approach to the thread assignment problem

In this section, we present a statistical method that overcomes the limitations of the approaches currently used for thread assignment in multithreaded processors. The analysis is comprised of three parts. First, we execute a sample of random thread 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 thread 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 thread assignments has good performance. Thirdly, we use Extreme Value Theory to statistically estimate the performance of the best thread 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 thread assignment method.

4.3.1 Finding thread 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 thread 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 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 [44]. 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: $P(A) + P(A') = 1$. 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 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, which is satisfied in the case of thread scheduling problems in multithreaded processors, as we have shown in Table 1.1.

Figure 4.2 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 Figure 4.2. First, that the probability asymptotically approaches 1 as the number of thread assign-

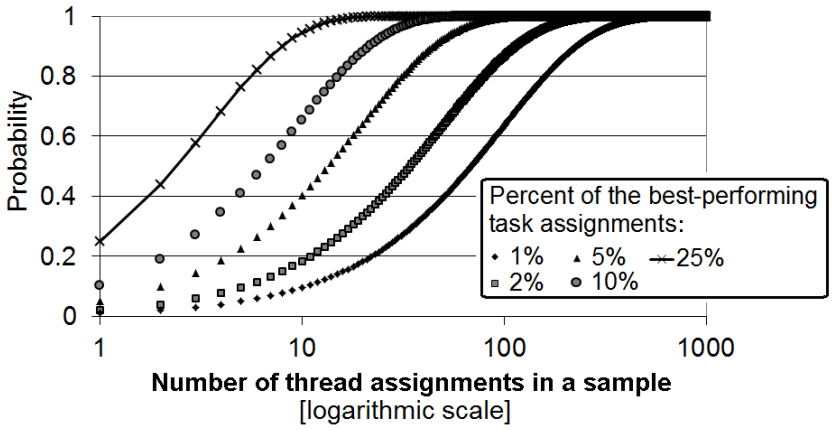


Figure 4.2: Probability that a sample contains a thread assignment from P% of the best-performing assignments

ments 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 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% 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 thread 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 thread assignments is sufficient to capture at least one assignment with a good performance.

4.3.2 Cumulative Distribution Function

One way to determine which portion of the population of all thread assignments exhibits a good performance is by plotting the Cumulative Distribution Function (CDF). We illustrate CDF with the following example. Figure 4.3 shows the CDF of all 1500 thread assignments for a network processing a workload of six threads. The details of the experimental framework are described in Chapter 2. 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 thread 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 thread assignment. The performance of the thread assignments ranges from 0.715 MPPS to 1.7 MPPS, meaning that

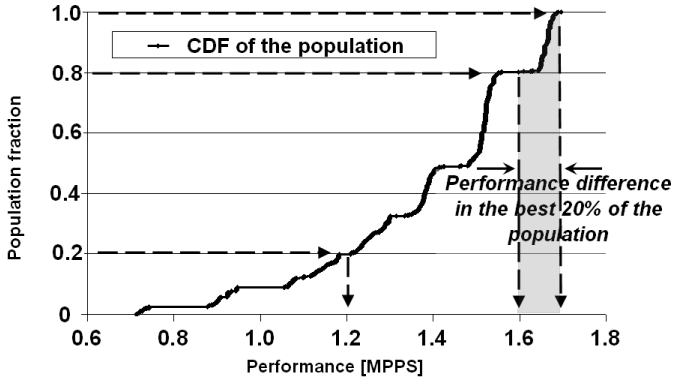


Figure 4.3: An example of Cumulative Distribution Function

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 thread assignments can be directly determined from the CDF of the population. For the data presented in Figure 4.3, the performance difference in 1% of the best-performing thread assignments [1] is very low, below 10,000 PPS which is only 0.6% of the optimal system performance. However, in general, since running all thread 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 thread assignments can be used to construct an Empirical CDF (ECDF) that estimates the CDF of the whole population [74, 88]. 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 thread 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 thread assignments, which are the main goals of our study.

4.3.3 Estimation of the optimal performance

One way to evaluate any thread assignment approach is to compare the performance of the thread 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 thread assignments is vast, the optimal system performance cannot be determined [87]. In this chapter, we propose using statistical inference methods to

[1] We refer to the performance difference (in MPPS) between population franction 1.0 and 0.99 respectively for the CDF presented in Figure 4.3.

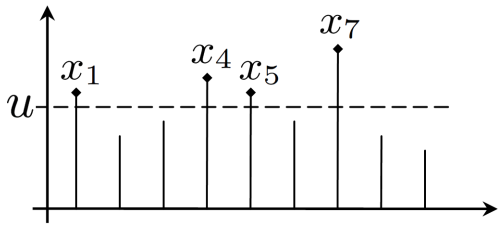


Figure 4.4: Exceedances over the threshold u

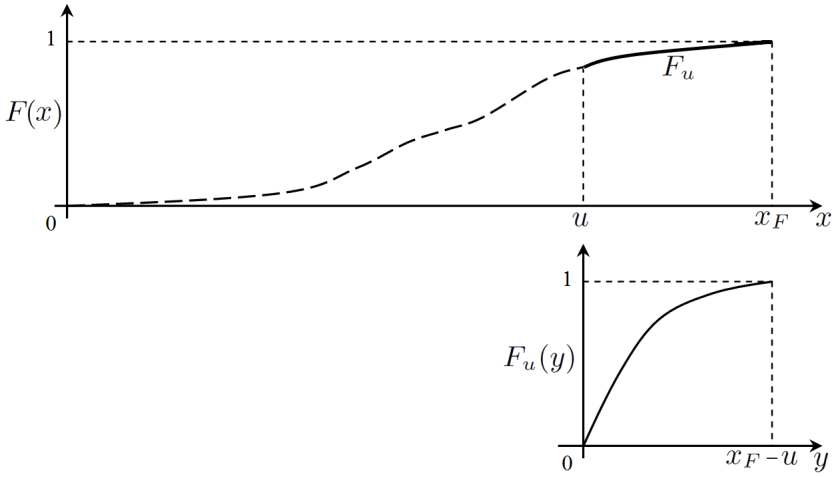


Figure 4.5: Cumulative distribution function $F(x)$ and and corresponding conditional excess distribution function $F_u(y)$

estimate the optimal system performance based on the measured performance of a sample of random thread assignments.

4.3.3.1 Extreme value theory

We estimate the performance of the best thread assignment using Extreme Value Theory (EVT). EVT is a branch of statistics that studies extreme deviations from the median of distributions [26, 34]. 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 4.4, 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

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

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 4.5 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 Pickands - Balkema - de Haan theorem [25, 123]:

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*

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

for $y \in [0, (x_F - u)]$ if $\xi \geq 0$ and $y \in [0, -\frac{\sigma}{\xi}]$ if $\xi < 0$, 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 4.3.3.2 in Step 2 and Step 3 of the presented analysis. GPD is defined with two parameters: shape parameter ξ and scaling parameter σ. 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 thread assignment) can be computed as $u - \frac{\sigma}{\xi}$, where σ and ξ are the GPD parameters and u is the selected threshold [70, 89].

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 [70, 89]. 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 ξ are always $\hat{\xi} < 0$. Therefore, for the sake of the simplicity of the presented mathematical formulas, in the rest of the chapter we do not present $G_{\xi,\sigma}(y)$ formulas for parameter $\xi = 0$.

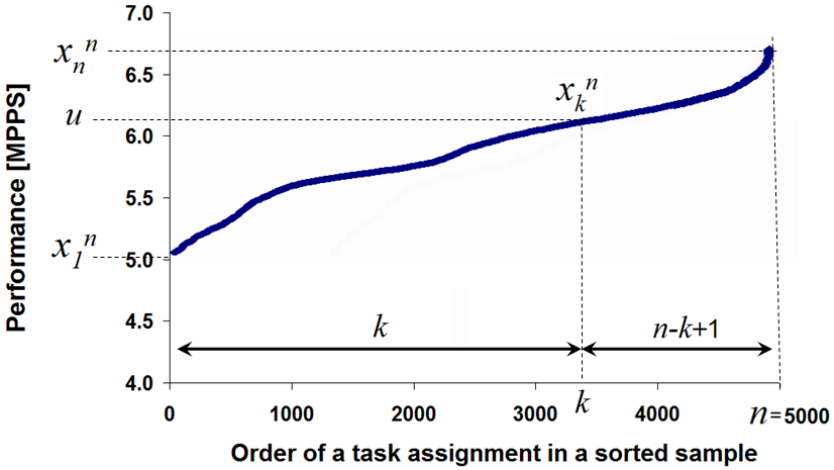
4.3.3.2 Application of Peak Over Threshold method

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

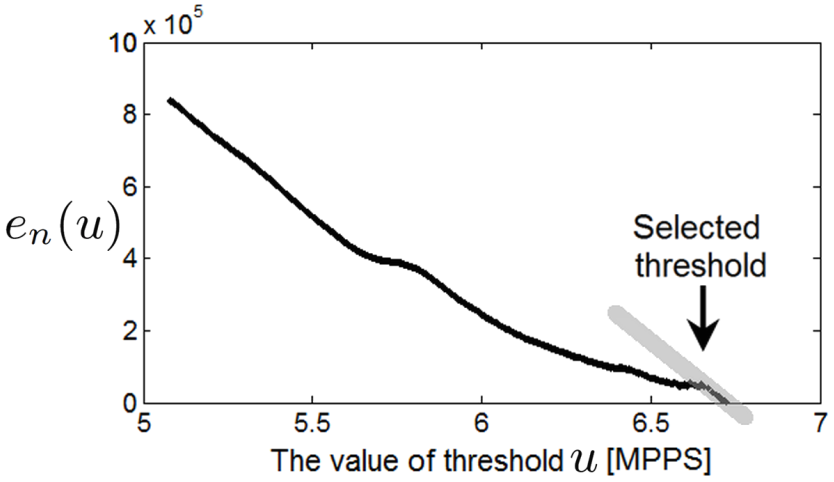
Step 1: Generate the sample of random thread assignments, execute the assignments on the target machine, and measure the performance of each assignment. A requisite of the presented statistical analysis is that the selected thread assignments in the sample are independent and identically distributed (*i.i.d.*). Intuitively, random variables are independent if knowing the value of one of them gives no new information about the values of the others; they are identically distributed if they all have the same probability distribution, which does not have to be uniform. Uniform distribution would mean that each kernel partition would be selected with the same probability. The sample assignments have to be taken from a single population using sampling with replacement [153]. The method we use to generate *i.i.d.* thread assignments is described next.

For example, assume that a workload of T threads 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 thread in the workload we randomly select an integer from this interval. The number assigned to each running thread represents the hardware context to which the thread is mapped to in a given thread assignment. After mapping all threads, we check if the generated thread assignment is valid. An assignment is not valid if two or more threads 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 thread 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* thread assignments.

Step 2: Select the threshold u. The selection of the threshold u is an important step in POT analysis. Gilli and Këllezi [70, 89] propose using *sample mean excess plot*, a graphical tool for threshold selection. This method first sorts all thread assignments in a sample in non-decreasing performance order: $x_1^n \leq x_2^n \leq \cdots \leq x_n^n$. Figure 4.6(a) shows the sorted performance of 5000 random thread assignments for a workload comprised of 24 threads of the *IPFwd-L1* application (see Section 2.3). 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



(a) Ordered sample of thread assignments



(b) Sample mean excess plot

Figure 4.6: Selection of the threshold u

mean excess function $e_n(u)$:

$e_n(u) = \frac{\sum_{i=k}^{n}(x_i^n - u)}{n-k+1}$, 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 4.6(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 ξ 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 4.6, the threshold should be selected to be around 6.6×10^6. 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 thread 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* [26, 89]. 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 chapter, 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 thread assignments in the whole sample. This is a commonly used limit in studies that use POT analysis [70, 89].

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 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 4.6(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 [35, 75, 81, 146]. 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 [23]. The GPD is defined with parameters ξ and σ. 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 ξ_0 and σ_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 ξ and σ 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 $\{y_1, y_2, ..., y_m\}$:

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

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

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 [41]. The values $\hat{\xi}$ and $\hat{\sigma}$ are called the point estimate of the parameters ξ and σ, respectively.

Step 4: Estimate the optimal system performance (the upper performance bound of

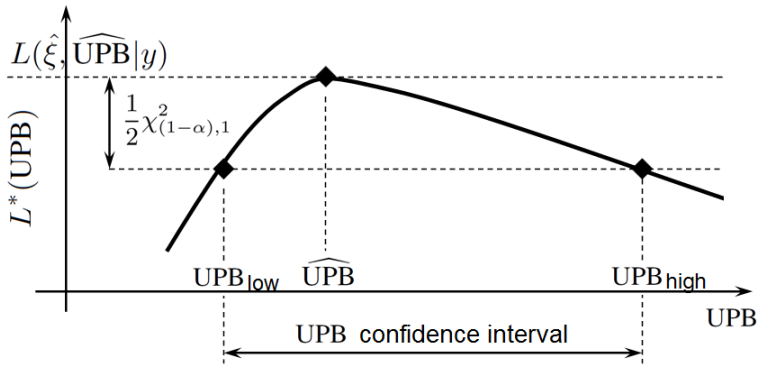


Figure 4.7: UPB confidence interval

all thread 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 chapter. 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* [23] which consists of the following steps:

(i) Define GPD as a function of ξ and UPB:

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

(ii) Determine the corresponding probability density function:

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

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

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

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

(iv) Find the UPB confidence interval. We determine the confidence interval for UPB using likelihood ratio test [23] and Wilks's theorem [42, 161, 162]. 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 (ξ 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 χ^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:

$$L(\hat{\xi}, \widehat{\text{UPB}}) - L^*(\text{UPB}) < \frac{1}{2}\chi^2_{(1-\alpha),1} \tag{4.1}$$

$\chi^2_{(1-\alpha),1}$ is the $(1-\alpha)$-level quantile of the χ^2 distribution with one degree of freedom ($df_1 - df_2 = 1$). α is the confidence level for which we compute UPB confidence intervals. We illustrate the computation of the UPB confidence interval in Figure 4.7. 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 4.1. 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.

4.3.4 Summary of the statistical analysis

In this section, we have presented two statistical methods. The first method computes the probability that a sample of n randomly selected thread 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 threads 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 thread 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 thread 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.

4.4 Results

In this section, we apply the presented statistical approach to the thread assignment of multithreaded network applications running on the UltraSPARC T2 processor. We start by analyzing whether a sample of random thread 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 thread assignment in the sample. Finally, we show how the presented analysis can be applied in the industrial case study for thread 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 threads.

4.4.1 Finding thread assignments with good performance

In Section 4.3.1, we presented a method to compute the probability that a sample of randomly selected thread assignments contains at least one of $P\%$ of best-performing assignments. We showed that samples containing more than several hundred random thread assignments capture at least one in 1% or 2% of the best performing thread 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 thread assingments asymptotically approaches 1 as the number of thread assignments in the sample exceeds 1000 (see Figure 4.2).

In order to analyze if further increasing the number of thread assignments in the sample increases the performance of the captured best-performing assignment, we executed experiments for 1000, 2000, and 5000 random thread assignments. The results of the experiment are plotted in Figure 4.8, in which the X axis lists different benchmarks and the Y axis shows the performance of the best thread assignment in the sample, measured in processed Packets Per Second (PPS). We observed that increasing the sample size from

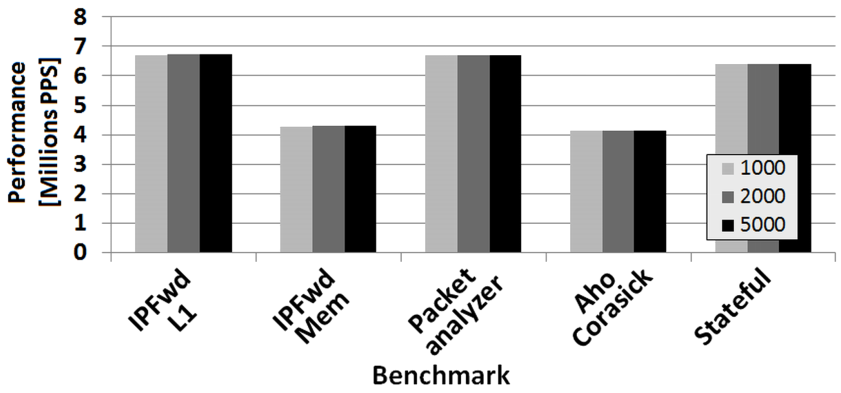


Figure 4.8: Performance of the best thread assignment in the random sample

1000 to 5000 only negligibly improves the performance of the captured best thread 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 thread assignment.

In order to analyze how good the performance of the captured best thread 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 thread assignment.

4.4.2 Estimation of the optimal performance

In Section 4.3.3, 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 thread assignment based on the measured performance of a sample of random assignments. Figure 4.9 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 thread 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

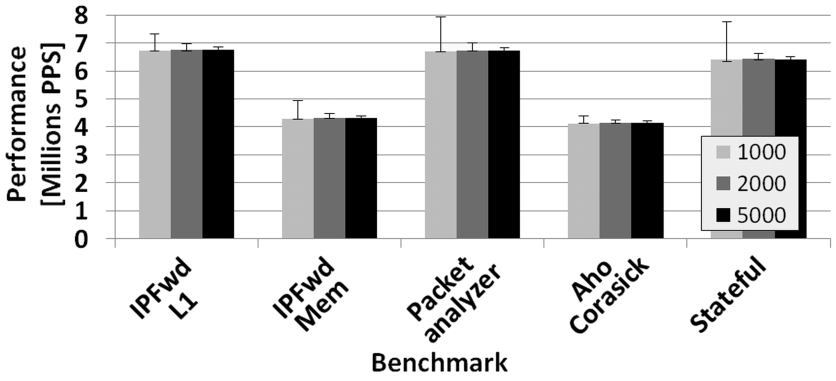


Figure 4.9: Estimated best-performing schedule performance

0.95 confidence level.

From the data presented in Figure 4.9, 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 thread assignments in the right tail of the cumulative distribution function (see Figure 4.5) are used to estimate the parameters of the Generalized Pareto Distribution (GPD). As we stated in Section 4.3.3.2, 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 thread assignments, respectively. As we increase the number of thread assignments in the sample, more observations can be used to estimate the optimal system performance, which, in turn, leads to more precise estimations.

Figure 4.10 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.

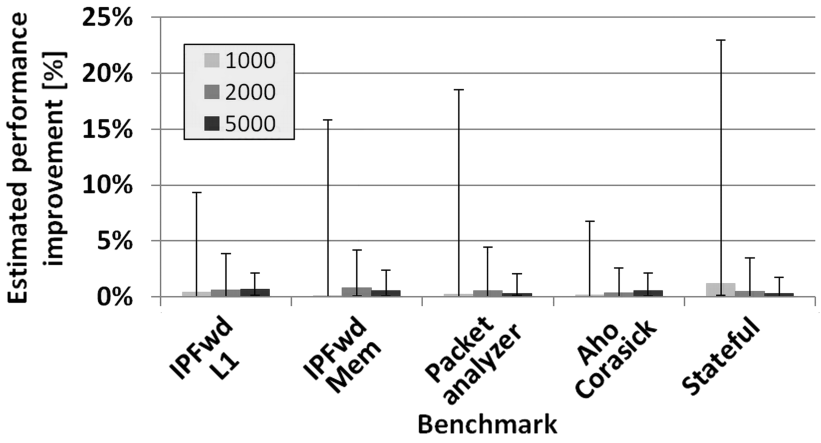


Figure 4.10: Estimated performance improvement

We reach several conclusions from the results that are presented in Figure 4.10. For 1000 thread 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%, respectively. 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 thread 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 4.3 to estimate the optimal system performance for a given workload. We used the estimated performance to understand how good the best thread assignments captured in the random samples are. In our experiments, running only several thousand random thread 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 thread assignment.

4.4.3 Case study

Next, we show how the presented analysis can be applied to an industrial case study for the thread assignment of network applications. In the presented scenario, the main objective is to satisfy the given performance requirement of the best thread assignment that is captured in the random sample. In this scenario, the customer requires that the performance difference between the captured best thread 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 4.11. 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 thread 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 thread 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 acceptable 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 thread 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 thread assignments in the random sample increases, the performance of the captured best assignment increases

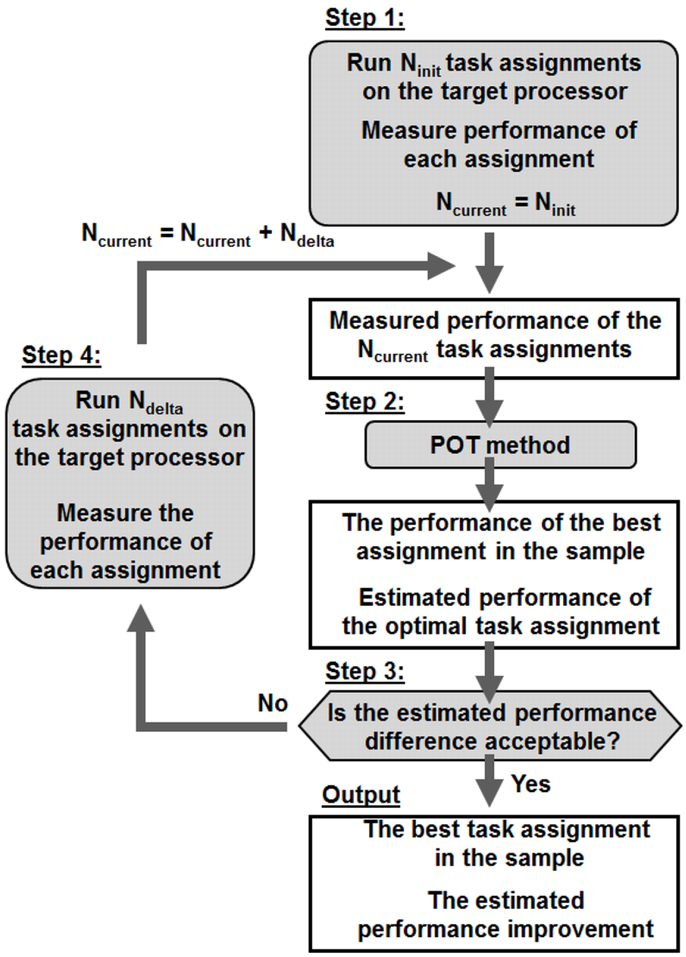


Figure 4.11: Case study: The schematic view of the algorithm

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 thread 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 $N_{init} = 1000$ thread 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 thread assignments in the sample needed to capture an assignment with the acceptable performance loss is presented in Figure 4.12. We present data for all five benchmarks in the suite

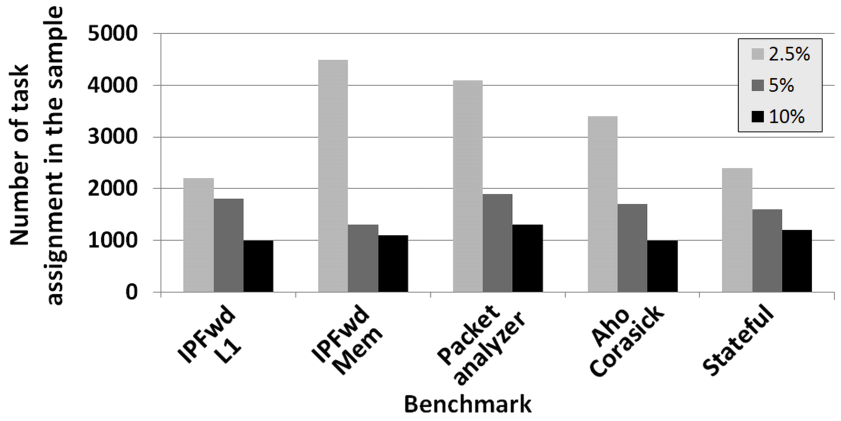


Figure 4.12: Case study: Required number of thread assignments

that are listed along the X axis. From the results presented in the figure we see that: (1) Running several thousand random thread 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 thread 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 thread assignments was sufficient to provide the required performance for all five benchmarks. (3) Required number of threads 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 thread 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.

4.4.4 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 thread assignments in the sample. In our target networking environment, as described in Section 2.4, around 1.5 seconds were enough to take a stable measurements of the performance of each thread assignment. Hence, the time needed to execute all experiments for samples of 1000, 2000, and 5000 thread 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 thread 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 thread 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 thread assignments. Performance predictors are models that estimate the performance of thread assignments based on the analysis of the target architecture and resource requirements of each thread in a workload (see Section 3.5). 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.

Application parameters and network traffic: It is very important that the behavior of an application under study in the sampling phase is representative of its run-time behavior. If the application can be configured by using different parameters, the parameters used to generate random samples have to correspond to the run-time parameters. Some of the parameters for the benchmarks used in the study are: size of the hash tables for the IPFwd benchmarks, the type of packet processing for the Packet Analyzer, the set of keywords that are searched in packet payload by the Aho-Corasick algorithm, or the size of the flow-record hash table used in the stateful network application. If the application behavior depends significantly on the network traffic, the traffic used in the sampling phase should be representative of the run-time traffic.

Workload selection: As we mentioned in Section 4.1, for processors with one level of resource sharing, the thread scheduling is done in a single step, called *workload selection*: out of all ready-to-run threads, the OS selects a set of threads (workload) that will concurrently execute on the processor [85]. As all threads share the same processor resources homogeneously, the way they interfere is independent of their distribution. In processors with several levels of resource sharing, thread scheduling requires an additional step, called *thread assignment*. Once the workload is selected, the threads 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 2.2). Hence, in

this kind of environment, the workload selection is not an issue and the optimal thread assignment is the only scheduling problem. In this chapter, we have shown how our statistical approach can be applied to the problem of the optimal thread 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 4.3.

4.5 Related work

Numerous studies address the problem of thread assignment of applications running on multithreaded processors, see Section 3.5. However, to the best of our knowledge, the work of Jiang et al. [87] is the only systematic study devoted to find the optimal thread assignment of applications running on multithreaded processors. First, the authors analyze the complexity of the thread assignment for multithreaded processors. Later, they propose several thread assignment algorithms. The authors use graphs to model interaction between simultaneously-running threads and use graph search to find the optimal solution. The main drawback of this study is that it assumes that the impact of thread interaction to system performance is known beforehand for all possible assignments, which is not satisfied in general 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.

4.6 Summary

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

In this chapter of the thesis, we proposed a statistical approach to the problem of optimal thread assignment. In particular, we showed that running a sample of several hundred or several thousand random thread 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 thread 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 thread assignment of multithreaded network applications running on the UltraSPARC T2 processor. Our results showed that running several thousand random thread assignments provided enough information for the precise estimation of the performance of the optimal thread 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.

Chapter 5

A statistical approach to kernel partitioning of streaming applications

One of the greatest challenges in computer architecture is how to write efficient, portable, and correct software for multicore multithreaded processors. A promising approach is to expose more parallelism to the compiler, through the use of domain-specific languages. The compiler can then perform complex transformations that the programmer would otherwise have had to do. Many important applications related to audio and video encoding, software radio and signal processing have regular behavior that can be represented using a stream programming language. When written in such a language, a portable stream program can be automatically mapped by the stream compiler onto multicore hardware. One of the most difficult tasks of the stream compiler is partitioning the stream program into software threads. The choice of partition significantly affects performance, but finding the optimal partition is an NP-complete problem.

In this chapter, we present a method, based on Extreme Value Theory (EVT), that statistically estimates the performance of the optimal partition. Knowing the optimal performance improves the evaluation of any partitioning algorithm, and it is the most important piece of information when deciding whether an existing algorithm should be enhanced. We use the method to evaluate a recently-published partitioning algorithm based on a heuristic. We further analyze how the statistical method is affected by the choice of sampling method, and we recommend how sampling should be done. Finally, since a heuristic-based algorithm may not always be available, the user may try to find a good partition by picking the best from a random sample. We analyze whether this approach is likely to find a good partition. To the best of our knowledge, this study is the first application of EVT to a graph partitioning problem.

5.1 Introduction

Stream programming languages represent the program as concurrent kernels, which communicate only via point-to-point streams. A kernel is a basic computation block with a user-defined function that processes input data streams into output data streams. Dependencies between different kernels are described explicitly through the communication data channels. The whole application can be represented as a *stream graph*. The nodes of the stream graph correspond to the kernels, while the directed edges represent the communication data channels.

There are three main advantages of stream programming languages, compared with traditional languages such as C. First, a stream programming language is a domain-specific language, which provides a natural way to describe streaming applications. Second, when the program is described using a stream language, the compiler may perform complex optimizations over the stream graph, producing an efficient multithreaded program. Some optimizations involve major changes to the program's structure and data layout. Third, unlike some other parallel programming models, including multithreading, a stream program is deterministic, and therefore easier to debug.

In order to optimally exploit the potential of high-end multithreded processors, stream programs are compiled into multithreaded executables by the stream compiler. One of the most important tasks of the compiler is to partition the kernels in the stream graph into software threads.

Kernel partitioning can significantly affect the overall system performance. For example, for the benchmarks included in the StreamIt 2.1.1 suite, the relative performance difference between good and bad kernel partitions of the same benchmark mapped into four software threads ranges from 2.4× to 3.9×, and on average it is 3.5×.

The optimal kernel partition cannot be determined because the essence of the analysis is graph partitioning, which is an NP-complete problem [63, 67]. Due to the large exploration space, brute force exploration is impractical: as streaming applications comprise tens or hundreds of interconnected kernels (54 kernels on average in the StreamIt 2.1.1 suite), the number of possible kernel partitions is vast. For example, the *channelvocoder* benchmark has 55 kernels, and it can be distributed into 10^{22} partitions of exactly four software threads. The number of possible kernel partitions increases rapidly with the number of output software threads, so the number of partitions using eight threads is 10^{34}.

As kernel partitioning is an intractable problem, it is impossible in general to know the performance of the optimal partition. Therefore, the room for improvement of the existing heuristic-based kernel partitioning algorithms is also unknown. It is hard to decide whether to invest additional effort to try to improve a given algorithm, since it may already be close to optimal.

In this chapter of the thesis, we present a statistical approach to the kernel partitioning problem. The main contributions of our study are:

- We present a method based on EVT that statistically estimates the performance of the optimal kernel partition. To the best of our knowledge, this is the first study that applies EVT to a graph partitioning problem.
- We show that the sampling method, used to generate random partitions, has a significant effect on the applicability of the statistical method. We analyze different sampling methods, and our results strongly recommend that the samples should be uniformly distributed.
- We use the estimates of the optimal performance to evaluate a state-of-the-art heuristic-based kernel partitioning algorithm.
- Finally, since a complex heuristic-based algorithm may not always be available, the user may pick the best from a random sample, and measure its quality using the estimates of the optimal performance. We analyze whether random sampling on its own is likely to find a good kernel partition.

The presented analysis is evaluated for the benchmarks included in the StreamIt 2.1.1 suite. The method based on EVT successfully estimates the performance of the optimal kernel partitions for all the benchmarks under study. In all experiments, the two different kernel partitioning methods, the heuristics-based algorithm and the method based on random sampling, detected kernel partitions with practically the same performance.

The rest of the chapter is organized as follows. Section 5.2 describes metrics that can be used to measure the performance of streaming applications. Section 5.3 discusses methods to generate random samples of kernel partitions. Section 5.4 presents the statistical analysis that we use to estimate the performance of the optimal kernel partition. In Section 5.5, we apply the presented analysis to the StreamIt 2.1.1 benchmark suite, and evaluate the results. Section 5.6 describes related work, and Section 5.7 presents the summary of the study.

5.2 Background

In this section, we describe different metrics that could be of interest when doing kernel partitioning. We also briefly describe related work with a focus on the conclusions that directly affect our study.

5.2.1 Target metric

There are several metrics that can describe the performance of streaming applications. In our study, we analyze the *cost* of streaming applications. The *cost* is proportional to the time needed to process a fixed amount of input data. This metric corresponds to execution time for non-streaming applications. Other metrics include energy or power, the hardware utilization of the target architecture, or some weighted sum of them.

The input to the statistical analysis is the cost of each kernel partition in a random sample. The costs are generated using metrics from the StreamIt 2.1.1 compiler. Instead of using the streaming compiler, the target metric could alternatively have been measured using real execution or simulation.

The proposed statistical approach and the general conclusions of this study are independent of both the target metric and the way in which the metric is evaluated: program compilation, execution or simulation. It is important to note, however, that the results from the statistical analysis are clearly dependent on the quality of the samples provided to it.

If the application behavior is sensitive to its input data, which is generally not the case for streaming applications, the user should consider the analysis for different input datasets that are representative for different application behavior.

If the user wants to use the statistical method for multiple objective functions separately, then it is only necessary to do a full set of compilations, executions or simulations once. After obtaining a complete set of metrics, the statistical analysis can be done multiple times using different metrics.

5.2.2 Convexity constraint

Carpenter et al. [32] present a partitioning and allocation algorithm for an iterative stream compiler. The algorithm produces kernel partitions that are easier to compile and that require short pipelines of software threads. The authors evaluate their proposal on the

benchmarks included in StreamIt 2.1.1 suite.

One of the conclusions in that study is that the kernel partition should be convex. A kernel partition is convex if the dependencies between different software threads form an *acyclic* graph. This means that every directed path between two kernels in the same software thread is internal to that thread. The reason for the convexity constraint is that the choice of partition affects the length of the pipeline generated by the streaming compiler. The convexity constraint controls the length of that pipeline. The authors demonstrate that without the convexity constraint, the compiler may generate long pipelines of software threads, which increases memory use and latency of the inter-thread communication, significantly affecting the overall performance.

We follow these instructions and focus our study on the analysis of *convex* kernel partitions. We pay special attention to generating random samples comprised only of kernel partitions that satisfy the convexity constraint. It is important to notice, however, that convexity is not a requirement of the proposed statistical approach. The approach can be also used for the analysis of non-convex kernel partitions.

Although the convexity constraint significantly reduces the number of kernel partitions, their number is still vast, and brute force exploration is impractical. For example, the fm benchmark can be distributed into 10^{12} convex kernel partitions of exactly four software threads, radar into 10^{14} partitions, filterbank into 10^{20}, and vocoder into 10^{23}.

5.3 Sampling methods

In order to apply Extreme Value Theory (EVT) to the kernel partitioning problem for streaming applications, we need to generate random convex partitions of the stream graph that are independent and identically distributed ($i.i.d.$). Intuitively, random variables are independent if knowing the value of one of them gives no new information about the values of the others; they are identically distributed if they all have the same probability distribution, which does not have to be uniform. Uniform distribution would mean that each kernel partition would be selected with the same probability.

The different methods we used to select the random $i.i.d.$ kernel partitions are described next. For each method, we describe how to select a single random kernel partition. To generate a sample of N $i.i.d.$ kernel partitions, repeat the sampling method N times.

5.3.1 Depth-First Search (DFS)

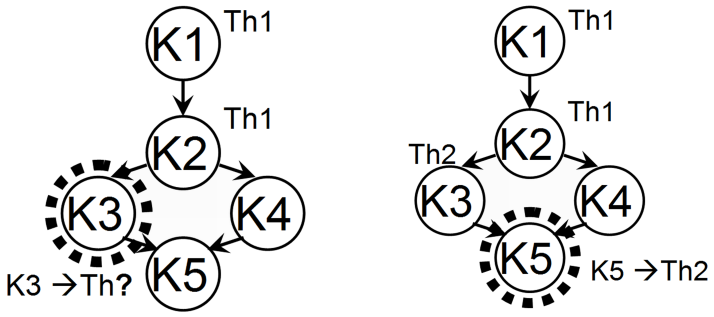
The first sampling method generates a random kernel partition using a depth-first search (DFS) of the stream graph. The kernels are visited in sequence, each kernel being assigned with equal probability to any software thread that would not violate the convexity constraint. Thus, this method generates random kernel partitions in a single traversal of the stream graph.

We illustrate the sampling method with the example shown in Figure 5.1. The example stream graph contains five kernels (K1 to K5) that are to be compiled into two software threads (Th1 and Th2). For example, assume that kernels K1 and K2 are already assigned to Th1, and that next kernel to be assigned is K3 (Figure 5.1(a)). K3 can be assigned with equal probability to Th1 or Th2. If K3 is assigned to Th2, kernel K5 has to be assigned to Th2 in order to generate a convex partition (see Figure 5.1(b)). Since the number of candidate threads that do not break the convexity constraint decreases rapidly, the DFS sampling method often generates kernel partitions with an unbalanced number of kernels per threads.

5.3.2 Edge Contraction (EC)

The second sampling method generates a random partition using edge contraction of the stream graph. Initially, each kernel is placed in its own cluster. Then, the edges of the stream graph are visited in random order. In each step the selected edge of the graph is contacted by fusing the clusters connected through this edge. If the resulting graph violates the convexity constraint, the contraction is undone. The process is continued, by moving to the next edge in random sequence, until the number of clusters equals the number of software threads. Finally, the clusters are randomly assigned to the threads.

Figure 5.2 illustrates the EC kernel partitioning of a simple stream graph into two software threads. For example, assume that K1→K2 edge is selected as the first edge to be contracted. In this case, kernels K1 and K2 are fused into a single cluster while the rest of the graph is not modified. Afterwards, we illustrate the contraction of edges K3→K5 and (K1&K2)→K4. Finally, the clusters are randomly distributed among software threads. In comparison with DFS, the EC sampling method generates random kernel partitions with a balanced number of kernels.



(a) Assigning kernel K3 (b) Assigning kernel K5

Figure 5.1: DFS sampling method

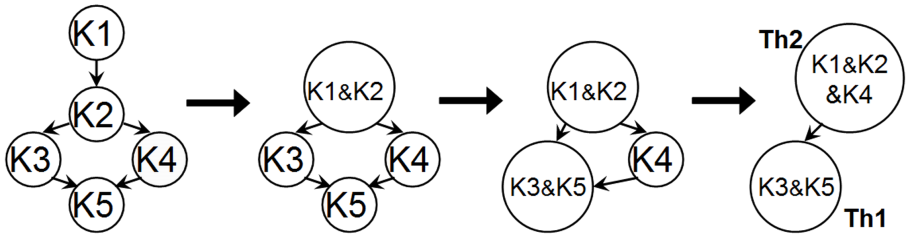


Figure 5.2: EC sampling method: Contracting K1→K2, K3→K5, and (K1&K2)→K4 edges, respectively.

5.3.3 Edge Contraction with Filter (EC-F)

The third sampling method is an enhancement of the EC method, designed to bias the sampling towards kernel partitions with a low cost, which will lead to good application performance. The EC-F method selects the edges of the stream graph in random order, fuses the corresponding clusters, and checks whether the convexity constraint is violated, as for the EC sampling method. The only difference is that EC-F performs an additional check: if contracting the current edge generates a cluster with a high cost (i.e. a cost that exceeds a given threshold), the contraction is undone and the process is repeated for a different edge. In the presented experiments, the threshold was the lowest cost detected in ten random kernel partitions generated using the EC sampling method.

It may happen that, although the number of clusters is still greater than the number of threads, none of the edges can be contracted without creating a cluster of cost exceeding the threshold. In this case, the remaining edges are visited in a new random order, and edges are contracted without checking whether the cost of the final clusters is over the threshold. Finally, the clusters are then randomly assigned to threads. In contrast with other presented sampling methods, this method does take into account the cost of

each particular kernel when generating partitions. The main target of this algorithm is to generate random partitions with a balanced cost among clusters.

5.3.4 Uniformly Distributed (UD) sampling

The final sampling method generates a uniform sampling distribution of the kernel partitions. This means that each convex kernel partition is selected with the same probability. It is important to notice that the statistical method used in the study does not require the kernel partitions to be uniformly distributed, since it only requires their cost to be independent and identically distributed ($i.i.d.$). In general, previous sampling methods do not provide uniformly distributed kernel partitions.

This sampling method comprises three steps.

Step 1: We analyze different kernel partitions using the *partition graph*, the graph of all possible convex partitions of the stream graph under study. Each node of the partition graph is a different kernel partition, so the number of nodes is equal to the number of partitions. There is an undirected edge between two nodes of the partition graph if they differ in the assignment of exactly one kernel partition. Also, each node contains a self-loop edge, an edge that connects the node to itself. Due to its large size (this is an NP-complete problem), the partition graph is never actually constructed in its entirety. An example partition graph, for a small stream program that is to be assigned to two software threads, is shown in Figure 5.3.

Step 2: We perform a random walk on the partition graph. First, we have to choose an initial node of the partition graph to start the random walk. This node can be selected by any method that generates random kernel partitions. In the experiments presented in our study, the initial kernel partition (initial node of the partition graph) is selected using the EC sampling method. The random walk starts from the initial node in the partition graph and calculates all its neighbors. Then it randomly chooses one of the neighbors (using a uniform distribution) to be the next node that is to be visited. The neighbor selection is repeated N times, with N large enough to potentially visit all the nodes of the partition graph of the benchmarks used in the study[1]. The last node of the partition graph that is visited is the outcome of the random walk. The probability that a given node is selected using the random walk is directly proportional to its degree (the number of its neighbors in the graph) [104]. As we know the probability of each visited node of the partition graph to be selected, the random walk generates samples with a known distribution.

[1] In our experiments, $N = 100$.

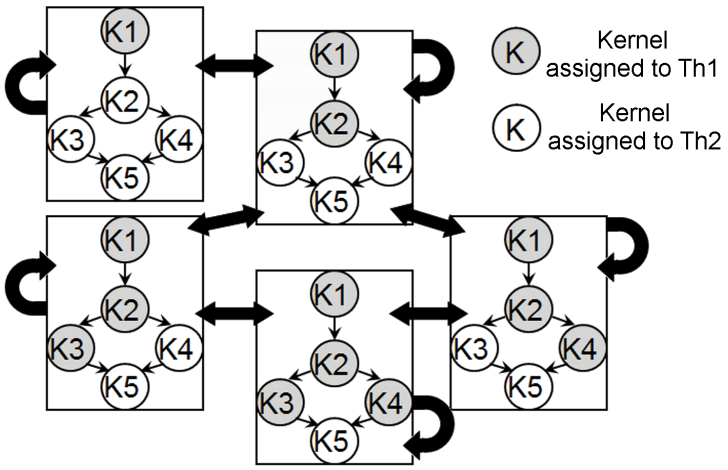


Figure 5.3: UD sampling method: Example partition graph for a small stream program

Step 3: In the final step of this sampling method, we convert the output of the random walk from a known distribution to a uniform distribution. In order to do so, each kernel partition selected using the random walk is included in the outcome of this sampling method with a probability that is inversely proportional to its degree in the partition graph. This way, every convex partition has the same probability of being selected, i.e. the method provides uniformly distributed samples.

5.3.5 Statistical *i.i.d.* tests

The sampling methods described in the previous section are designed to generate random $i.i.d.$ samples. After generating the samples, we perform statistical tests to confirm that they are indeed independent and identically distributed.

Wald–Wolfowitz test: The *Wald–Wolfowitz* test or *runs* test examines whether the observations in the sample are mutually independent [29, 48]. The test comprises two main steps. First, the costs of kernel partitions (non-negative real numbers) have to be converted into binary values. We converted the cost of a given kernel partition to *'0'* if its value was below the median cost in the sample, and converted it to *'1'* otherwise. This way, the sequence of non-negative real numbers was converted into a sequence of *0s* and *1s*, e.g. *000110000*. In the second step, the test analyzes the sub-sequences of consecutive identical values (*0s* or *1s*), which are referred to as $runs$. For example, the sequence *000110000* is composed of three runs: *000*, *11*, and *0000*. The Wald–Wolfowitz test validates that the observations in the sample under study are mutually independent if the lengths of the runs follow a Gaussian distribution [29]. The mutual independence

hypothesis was tested at the 0.05 significance level. All the samples used in the study passed the test.

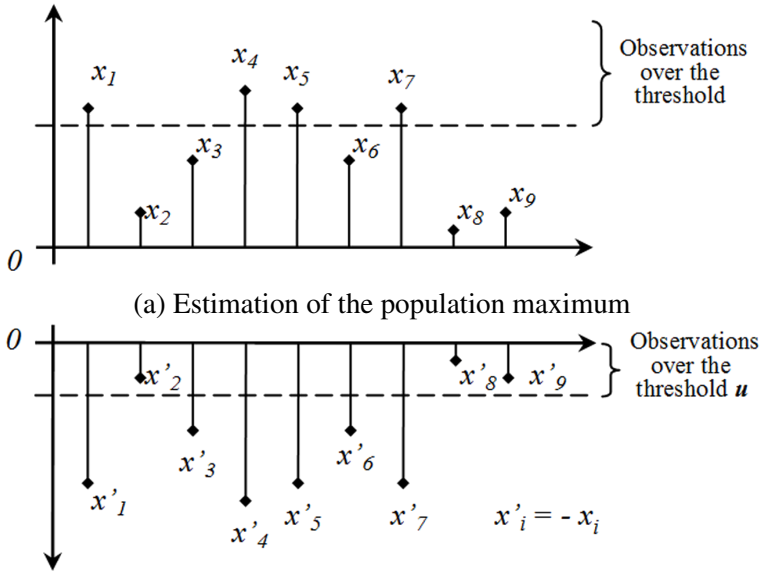
Kolmogorov–Smirnov test: In order to validate that selected kernel partitions in a given sample are identically distributed, we used a two-sample *Kolmogorov–Smirnov* test [48, 65]. The test compares the empirical cumulative distribution functions (ECDF) of two data sets and, based on the maximum distance between the two ECDFs, it confirms or rejects the hypothesis that the data sets correspond to the same distribution. The identically distributed test that we performed contains three steps. First, we generated a random sample of 20,000 kernel partitions and observed the cost of each partition. The costs of the kernel partitions in the sample followed the order in which the partitions were generated. Second, in each experiment, we observed two randomly-selected segments of m consecutive values from the original sample. Finally, we used a two-sample *Kolmogorov–Smirnov* test to check whether the randomly selected segments of the sample have the same probability distribution. If the kernel partition costs are indeed identically distributed, then all segments of consecutive values in the sample have the same distribution. For each sample used the study, we performed the test for segments of m =100, 500, 1,000 and 5,000 observations. All the samples used in the study passed the test at the 0.05 significance level.

5.4 A statistical approach to kernel partitioning of streaming applications

We estimate the minimal cost of kernel partitions (that lead to optimal performance) using Extreme Value Theory (EVT). EVT is a branch of statistics that studies extreme deviations from the median [26, 34]. One of the approaches in EVT is the *Peak Over Threshold* (POT) method. In its original form, the POT method takes into account only the distribution of the observations that exceed a given (high) threshold to estimate the population maximum [25, 123]. For example, in Figure 5.4(a), the observations x_1, x_4, x_5, and x_7 exceed the threshold and constitute extreme values, which can be used by POT analysis.

The POT method can be used also to estimate the population minimum [70, 89]. Estimation of the minimum requires the following five steps explained in detail in the next section:

- Obtain *i.i.d.* observations x_i of the cost of kernel partitions.



(a) Estimation of the population maximum

(b) Estimation of the population minimum

Figure 5.4: Exceedances over the threshold

- Invert the sign of the observations: $x'_i = -x_i$.
- Determine the threshold u, shown in Figure 5.4(b).
- Use the values x'_i over the threshold u to estimate the maximum cost of the inverse population ($Max(Cost_{Inv})$).
- The minimum cost of the original population ($Min(Cost)$) corresponds to the negative value of the maximum of the inverse population: $Min(Cost) = -Max(Cost_{Inv})$;

As we explained in Section 4.3.3, the POT method can also be explained using cumulative distribution functions (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

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

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.

The POT method is based on the Pickands - Balkema - de Haan theorem [25, 123]:

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

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

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 for numerous distributions that present real-life problems, F_u can be approximated with a GPD. For each particular problem, the decision of whether GPD can be used to model the problem or not, is based on how well the sample of observations can be fitted to GPD. We describe the goodness of fit of observations to GPD in Steps 3 and 4 of the analysis presented in Section 5.4.1. GPD is defined with two parameters: shape parameter ξ and scaling parameter σ. One of the characteristics of GPD is that for $\xi < 0$ the upper bound of the observed value equals $u - \frac{\sigma}{\xi}$, where σ and ξ are the GPD parameters and u is the selected threshold [70, 89].

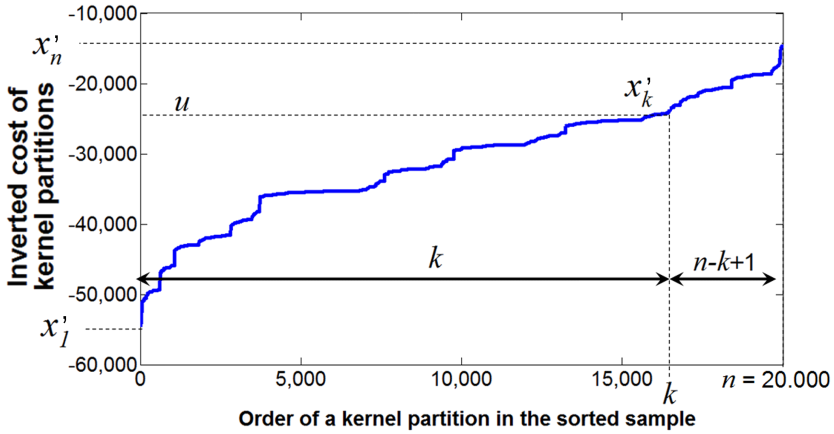
In Theorem 1, the definition of $G_{\xi,\sigma}(y)$ for $\xi = 0$ can only be used to model problems with an infinite upper bound [70, 89]. In this study we use the GPD to estimate the minimal cost of kernel partitions for streaming applications. The value of this cost is always finite, and the estimated values of the parameter ξ are always $\hat{\xi} < 0$. Therefore, for the sake of the simplicity of the presented mathematical formulas, in the rest of the thesis we do not present $G_{\xi,\sigma}(y)$ formulas for $\xi = 0$.

5.4.1 Application of Peak Over Threshold method

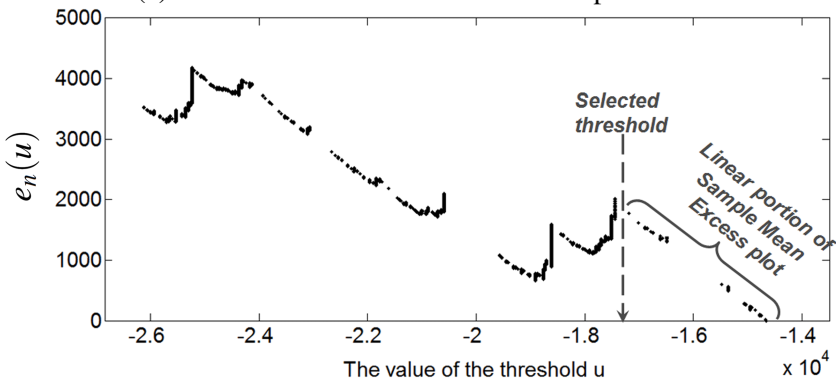
We use the POT method to estimate the minimal cost of kernel partitions for streaming benchmarks based on the cost of a sample of random partitions. Application of the POT method involves the following six steps:

Step 1: Generate the sample of random kernel partitions, and determine the cost of each partition in the sample (x_i). A requisite of the presented statistical analysis is that the selected kernel partition must be independent and identically distributed (*i.i.d.*).The proposed methods to generate *i.i.d.* kernel partitions are described in Section 5.3. All of the samples used in the study passed the described *i.i.d.* tests.

Step 2: Invert the sign of the values of the observed costs. Originally, the POT method was used to estimate the maximum of a population based on a set of random *i.i.d.* obser-



(a) Ordered costs of the random kernel partitions



(b) Sample mean excess plot

Figure 5.5: Selection of the threshold for *mpeg2-subset*

vations. In order to estimate the *minimum* cost of kernel partitions, we invert the sign of the observed values ($x'_i = -x_i$) and estimate the *maximum* of the *inverse* population.

Step 3: Select the threshold u. The selection of the threshold u is an important step in POT analysis. Gilli and Këllezi [70, 89] propose using the *sample mean excess plot*, a graphical tool for threshold selection. This method first sorts all kernel partitions in the sample in non-decreasing cost order: $x'_1 \leq x'_2 \leq \cdots \leq x'_n$. Figure 5.5(a) shows the sorted cost of 20,000 uniformly distributed random kernel partitions of *mpeg2-subset* benchmark.

Then, the possible threshold u takes the values from x'_1 to x'_n ($x'_1 \leq u \leq x'_n$), and for each value we compute the sample mean excess function $e_n(u)$:

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

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

$x'_1 \leq u \leq x'_n$. Figure 5.5(b) shows the example of the sample mean excess plot for the *mpeg2-subset* benchmark.

As commented before, the estimated parameter ξ of GPD must be negative ($\hat{\xi} < 0$) to obtain the upper bound of the $Max(Cost_{Inv})$. A characteristic of the GPD with parameter $\xi < 0$ is that it has a 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 5.5, the threshold should be selected to be $u = -17,500$. The sample mean excess plot is also a 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 kernel partitions 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 GPD is a *quantile plot* [26, 89]. In a quantile plot, the sample quantiles x'_i 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.

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 kernel partitions in the whole sample. This is a commonly used limit in studies that use POT analysis [70, 89, 130].

Step 4: 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 fitted to GPD, and the parameters of the distribution can be estimated. For the sake of simplicity, we assume that observations from x'_k to x'_n in the sorted sample presented in Figure 5.5(a) exceed the threshold. We rename the exceedances $y_{i-k+1} = x'_i - u$ 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 [35, 75, 81, 146]. In our study, we used estimation based on the *likelihood*

function [23]. The GPD has parameters ξ and σ. The likelihood that a set of observations $Y = \{y_1, y_2, ..., y_m\}$ is the outcome of a GPD with parameters $\xi = \xi_0$ and $\sigma = \sigma_0$ is defined to be the probability that GPD with parameters ξ_0 and σ_0 has outcome Y.

We make use of 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 computations. The estimation of parameters ξ and σ 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 $\{y_1, y_2, ..., y_m\}$:

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

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

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 [41]. The values $\hat{\xi}$ and $\hat{\sigma}$ are called the point estimates of the parameters ξ and σ, respectively.

Step 5: Estimate the maximum of the inversed costs. The maximum of the inverse cost can be determined only for $\hat{\xi} < 0$. The point estimate of $Max(Cost_{Inv})$ is computed

as $\widehat{Max(Cost_{Inv})} = u - \hat{\sigma}/\hat{\xi}$.

In order to indicate the confidence of the estimate, we compute the confidence intervals of the estimated $\widehat{Max(Cost_{Inv})}$. The confidence intervals is computed using the *likelihood ratio test* [23], which consists of the following steps:

(i) Define GPD as a function of ξ and $Max(Cost_{Inv})$:

$$G_{\xi,Max(Cost_{Inv})}(y) = 1 - (1 - \frac{1}{Max(Cost_{Inv})-u}y)^{-1/\xi}$$

(ii) Determine the corresponding probability density function:

$$g_{\xi,Max(Cost_{Inv})}(y) = \frac{\partial G_{\xi,Max(Cost_{Inv})}(y)}{\partial y} = \\ = -\frac{1}{\xi(Max(Cost_{Inv})-u)}(1 - \frac{1}{Max(Cost_{Inv})-u}y)^{-\frac{1}{\xi}-1}$$

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

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

$$L(\xi, Max(Cost_{Inv})|y) = -n\ log(-\xi(Max(Cost_{Inv}) - u)) - \\ -(1+\tfrac{1}{\xi})\sum_{i=1}^{n} log(1 - \frac{1}{Max(Cost_{Inv}) - u}y_i)$$

(iv) Find the $Max(Cost_{Inv})$ confidence interval. We determine the confidence interval for $Max(Cost_{Inv})$ using the likelihood ratio test [23] and Wilks's theorem [42, 161, 162]. The maximum log-likelihood function is determined as

$$L(\hat{\xi}, \widehat{Max(Cost_{Inv})}|y) = \max_{\xi,Max(Cost_{Inv})}(L(\xi, Max(Cost_{Inv}))).$$

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

$$L^*(Max(Cost_{Inv})) = \max_{\xi}\ L(\xi, Max(Cost_{Inv})).$$

The function $L^*(Max(Cost_{Inv}))$ 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, the distribution of $2(L(\hat{\xi}, \widehat{Max(Cost_{Inv})}) - L^*(Max(Cost_{Inv})))$ converges to a χ^2 distribution with $df_1 - df_2$ degrees of freedom. Therefore, the confidence interval of $Max(Cost_{Inv})$ includes all values of $Max(Cost_{Inv})$ that satisfy the following condition:

$$L(\hat{\xi}, \widehat{Max(Cost_{Inv})}) - L^*(Max(Cost_{Inv})) < \frac{1}{2}\chi^2_{(1-\alpha),1} \tag{5.1}$$

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

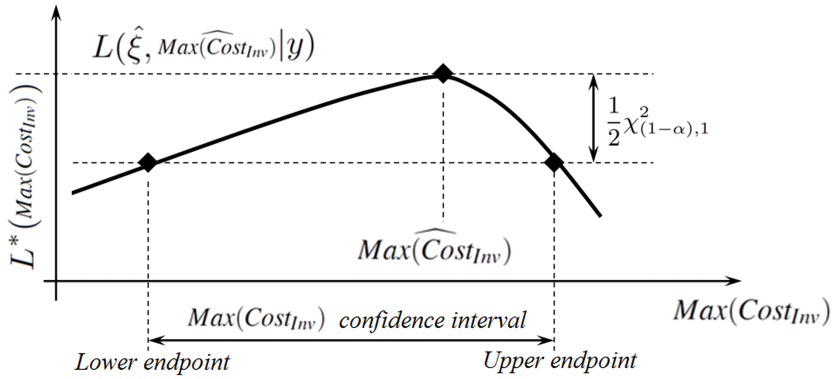


Figure 5.6: $Max(Cost_{Inv})$ confidence interval

dence intervals. We illustrate the computation of the $Max(Cost_{Inv})$ confidence interval in Figure 5.6. The figure plots $L^*(Max(Cost_{Inv}))$ for different values of $Max(Cost_{Inv})$. For $Max(Cost_{Inv}) = \widehat{Max(Cost_{Inv})}$, L^* reaches its maximum. The confidence interval of $Max(Cost_{Inv})$ includes all values of $Max(Cost_{Inv})$ that satisfy the condition $L^*(Max(Cost_{Inv})) > L(\hat{\xi}, \widehat{Max(Cost_{Inv})}) - \frac{1}{2}\chi^2_{(1-\alpha),1}$, which corresponds to Equation 5.1. We computed the $Max(Cost_{Inv})$ confidence interval using an iterative method based on the *fminsearch()* function included in Matlab® R2007a.

Step 6: Estimate the minimum cost of the kernel partitions. The minimum cost of the kernel partitions corresponds to the estimated maximum of the inverse cost:

$$Min(Cost) = -Max(Cost_{Inv}).$$

Also, the lower and upper endpoint of the $Min(Cost)$ confidence interval correspond to the inverse upper and lower endpoint of the $Max(Cost_{Inv})$ confidence interval, respectively.

The code that performs statistical *i.i.d.* test, generates the sample mean excess plots, infers the parameters of the GPD distribution, and estimates the minimum cost of kernel partitions was developed in Matlab® R2007a.

5.5 Results

In this section, we use the POT method to estimate, for each of the StreamIt 2.1.1 benchmarks, the cost of the optimal kernel partition. We compare the four sampling methods described in Section 5.3. We also evaluate the number of random kernel partitions that are required by the presented statistical approach. Finally, we analyze whether a good kernel

partition would be found using random sampling on its own.

Before using any heuristics-based algorithm for the concrete application under study, the user should check whether exhaustive search would be impractical. In general, the number of valid kernel partitions is vast (e.g. 10^{20}). It is possible, however, that a particular benchmark has a small stream graph, so that exhaustive search would work. For example, the *dct* benchmark included in the StreamIt 2.1.1 suite contains only eight kernels, linked in a simple stream graph. The number of partitions of this benchmark, onto four threads, is just 32. In this case, exhaustive search is the simplest and fastest way to find the optimal kernel partition.

5.5.1 Estimation of the minimal cost using the POT method

We apply our technique to estimate the performance of the optimal kernel partition on four threads, for each of the benchmarks in the StreamIt 2.1.1 suite. As discussed in the previous section, the *dct* benchmark can be solved using exhaustive search, leaving eleven benchmarks to be analyzed using the POT statistical method.

For each benchmark, we use the four sampling methods described in Section 5.3 to generate random samples, and use these samples as the input to the statistical analysis. Each sample contains 20,000 random kernel partitions. Table 5.1 shows whether or not the statistical method could produce an estimate of the optimal performance. Each row in the table corresponds to one of the benchmarks, and each column corresponds to a different sampling method. A tick sign (✔) means that the POT method did generate an estimate. An *NA* (Not Applicable) entry means that the statistical method failed to produce any estimate.

There are two reasons why the POT method is sometimes unable to produce an estimate. First, the lower bound of the estimated minimal cost may diverge to minus infinity. Second, the iterative method that determines the confidence bounds of the estimated minimal cost (see Step 5 in Section 5.4.1) may not converge to a solution. In all experiments in which the POT method was not applicable, the sample mean excess plot and the quantile plot strongly suggested that the POT method could not be applied to that dataset (see Step 3 in Section 5.4.1).

We see, from the results in the table, that the POT method, using the Depth First Search (DFS) sampling method, was successfully applied for only three out of eleven benchmarks. The results for the Edge Contraction (EC) method are better, but still moderate: the POT method estimated the minimum cost for seven out of eleven benchmarks.

Table 5.1: Applicability of the POT method

Benchmark	Sampling method			
	DFS	EC	EC-F	UD
bitonic-sort	NA	✔	NA	✔
channelvocoder	✔	NA	NA	✔
des	NA	✔	✔	✔
fft	NA	✔	NA	✔
filterbank	NA	NA	NA	✔
fm	✔	NA	NA	✔
mpeg2-subset	NA	✔	NA	✔
radar	✔	NA	NA	✔
serpent_full	NA	✔	NA	✔
tde_pp	NA	✔	NA	✔
vocoder	NA	✔	NA	✔

The Edge Contraction with Filter (EC-F) method was an attempt to improve load-balance over EC. However, the POT analysis could now only be applied to one of the benchmarks. We compared the costs of the random kernel partitions sampled by the EC and EC-F methods, and confirmed that the EC-F method did indeed select kernel partitions with lower cost. This was, however, not sufficient to make the samples appropriate for POT analysis. In future work, we plan to analyze this phenomenon in detail. Finally, when the POT method was applied to the uniformly distributed random samples (UD column of the table), a minimum cost was generated for all eleven benchmarks under study.

From the results presented in Table 5.1, we conclude that the sampling method is an important step in the analysis. All presented sampling methods select $i.i.d.$ samples and fulfill the requirements of the POT statistical analysis. However, only the uniformly distributed samples always led to an estimate of the cost of the optimal kernel partition. Other sampling method used to address *the same problem, for the same benchmarks, using the same statistical analysis* provided moderate (EC method) or low performance (DFS and EC-F methods).

5.5.2 Precision of the estimation

The results that we use to analyze the precision of the estimated values are presented in Figure 5.7. The X-axis of the figure lists the benchmarks, while the Y-axis shows the estimated minimal cost, i.e. the estimated cost of the optimal kernel partition. The results are presented relative to the cost of the best kernel partition captured in 80,000 random

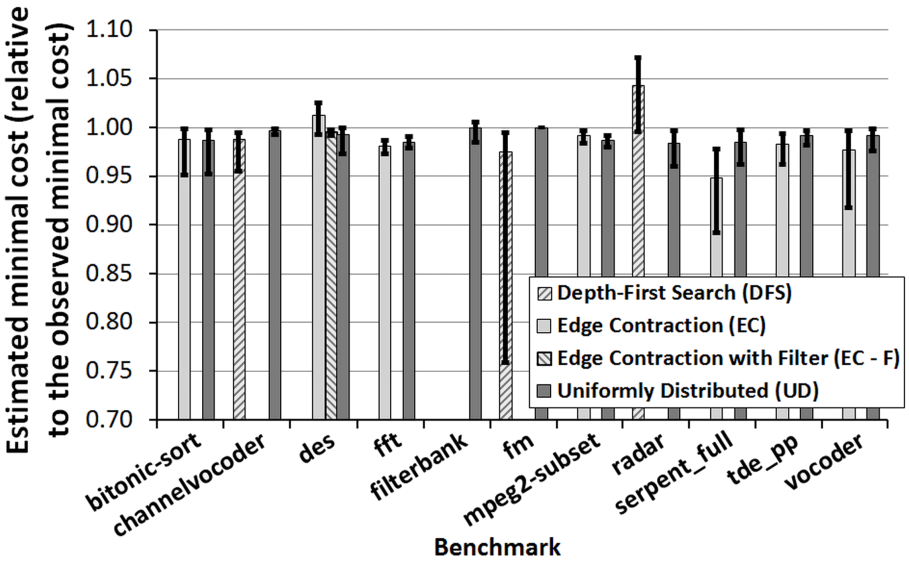


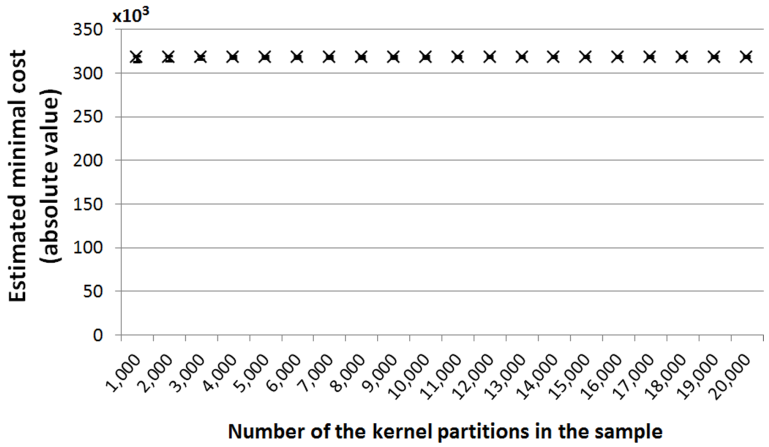
Figure 5.7: Estimated minimal cost

kernel partitions from all four sampling methods. Kernel partitions were generated using four different sampling methods (DFS, EC, EC-F, and UD), and each method generated 20,000 random partitions.

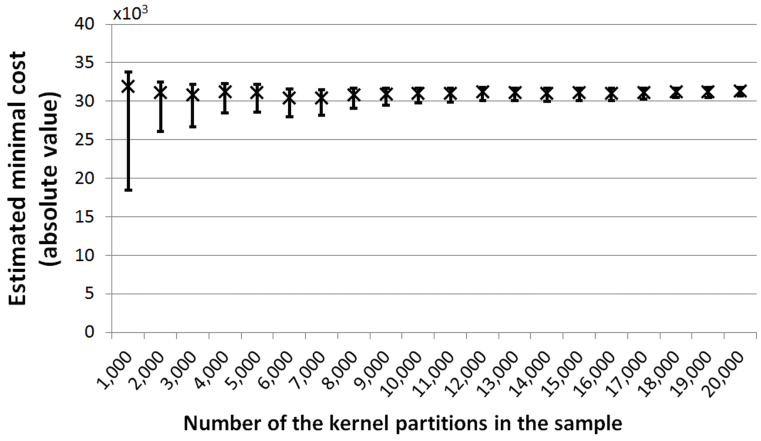
Different bars of the chart correspond to different sampling methods used: DFS, EC, EC-F, and UD. If the POT method could not be used to estimate the minimal cost for a given benchmark and a sampling method, the corresponding bar is not plotted. The height of the solid bars correspond to the point estimation of the minimal cost, while the error bars correspond to the confidence bounds for 0.95 confidence level.

High precision of the estimated minimal cost is indicated with the tight confidence bounds. The width of the confidence bounds is below 10% for all except one bar (DFS sample for the *fm* benchmark). For 18 out of 22 cases, the confidence bounds width is below 5%.

Required number of random kernel partitions: UD method is the only sampling method that provided samples appropriate for the POT statistical analysis for all the benchmarks under study. Therefore, from this point on, we analyze only the samples that are generated with this method. In order to understand the impact of the sample size on the estimated minimal cost, we generated samples that contain between 1,000 and 20,000 random kernel partitions. For each sample, we used the POT method to estimate the minimal kernel cost. Intuitively, we expect that the POT method provides more precise estimation as the number of kernel partitions in the sample increases. In general, larger



(a) channelvocoder benchmark (fastest convergence)



(b) serpent_full benchmark (slowest convergence)

Figure 5.8: The impact of the sample size on the estimation of the minimal cost (UD sampling method)

samples contain more kernel partitions in the tail that are fitted to the Generalized Pareto Distribution (GPD), and therefore the estimated GPD parameters and the minimal cost are more precise. Figure 5.8 shows the results for the *channelvocoder* and *serpent_full* benchmarks. In each figure, X-axis lists the number of random kernel partitions in the sample, while the Y-axis shows the estimated minimal cost. The cross markers show the point estimation of the minimal cost, and the error bars correspond to the confidence bounds for the 0.95 confidence level.

For the *channelvocoder* benchmark, 1,000 random kernel partitions are sufficient to estimate the minimal cost with a high precision (see Figure 5.8(a)). We detect similar results for *fft, filterbank, fm, mpeg2-subset, tde-pp,* and *vocoder*. On the other hand, for

the *serpent_full* benchmark, estimation based on 1,000 random kernel partitions has wide confidence bounds (see Figure 5.8(b)). Precise estimation of the minimal cost requires more than 8,000 random kernel partitions. The width of the confidence bounds reduces significantly as the sample increases from 1,000 to 8,000 kernel partitions. Further increment in the sample size only slightly improves the precision of the estimation. From the results shown in Figure 5.8(b), we also see that, as the sample size increases, the point estimation remains roughly the same and the confidence bounds converge to this value. Results for the benchmarks *bitonic-sort, des,* and *radar* follow the same trend.

Based on the presented analysis, we see that the sample size required for the precise estimation of the minimal cost significantly depends on the benchmark under study. If a user requires a minimal cost to be estimated with a given precision, we propose the following iterative method. The user can generate a small sample of random kernel partitions and estimate the minimal cost using the POT method. As long as the estimated cost does not fulfill the user's requirements, the user can increase the sample size and repeat the analysis.

5.5.3 Accuracy of the estimation

In general, the kernel partitioning problem is an intractable problem. However, for *bitonic-sort, des, fft, mpeg2-subset, serpent_full,* and *tde_pp* benchmarks partitioned into exactly four software threads, brute force exploration is time consuming, but feasible. Therefore, we were able to determine the cost of all the kernel partitions, and to compare the actual and the estimated best kernel partition costs for these benchmarks. The results for the *serpent_full* benchmark are shown in Figure 5.9. We also analyze the estimation accuracy for different numbers of uniformly distributed random kernel partitions in the sample. The X-axis of the figure lists the size of the sample, while the Y-axis shows the absolute value of the kernel partition cost. The cross data markers show the point estimation of the minimal cost, and the error bars correspond to the confidence bounds for the 0.95 confidence level. The actual best kernel partition cost is marked with the horizontal dashed line. Finally, we also plot the minimal kernel cost observed in each random sample (diamond data markers).

First, we observe that the estimated best kernel partition cost (with confidence bounds included) is always lower than the minimal kernel cost detected in the corresponding random sample. Intuitively, this is because the statistical method estimates that the best kernel partition cost in the population (out of all possible partitions) is not higher than the

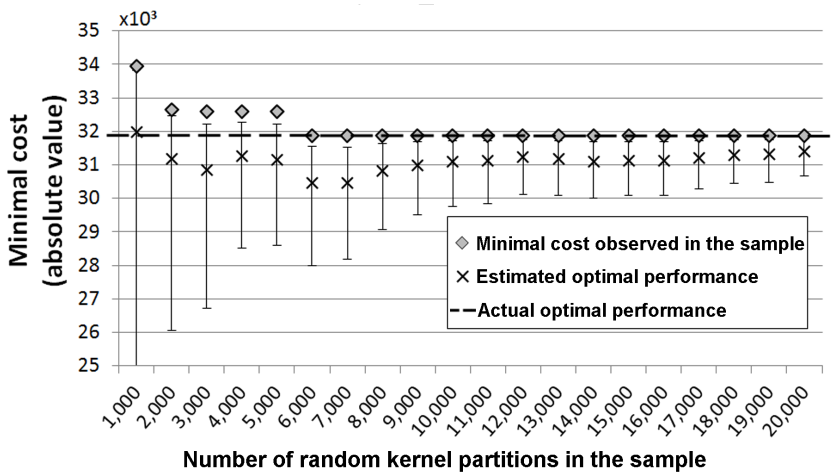


Figure 5.9: Comparison between the actual and the estimated kernel partition costs (serpent_full benchmark, UD sampling method)

minimal cost observed in the sample. We also detected that the upper confidence bound of the estimated best kernel partition cost asymptotically approaches the minimal kernel partition cost observed in the sample, as the confidence level of the estimation increases.

The estimation of the best kernel partition cost is accurate if its confidence bounds include the actual best (minimal) cost, which is satisfied for the samples that contain from 1,000 to 5,000 random kernel partitions in the Figure 5.9. For the samples that contain more than 6,000 kernel partitions, the presented statistical method slightly underestimates the minimal cost. This is because these samples capture a kernel partition with the best actual cost, as we explain in the previous paragraph. The underestimation is very low and it decreases with the number of kernel partitions in the sample, from 0.9% (6,000 kernel partitions) to 0.3% (20,000 partitions). The results for *bitonic-sort, des, fft, mpeg2-subset,* and *tde_pp* benchmarks follow the same trend.

Brute force exploration of the kernel partitioning problem for *channelvocoder, filterbank, fm, radar,* and *vocoder* benchmarks is infeasible. Therefore, for these benchmarks, we cannot determine the optimal kernel partition and its cost, and we cannot validate that the estimated values of the POT method were correct. However, from the results presented in Figure 5.7, we can detect that the estimation is incorrect if:

- The confidence bounds of different bars that correspond to the same benchmark do not overlap. This means that the POT method applied to different samples of the same benchmark estimated different minimal cost.

- The ratio between the estimated minimal cost (with confidence bounds included) and the minimal cost detected in random samples is higher than 1. This means that we detected a kernel partition with the cost that is lower than the estimated minimal cost.

From the results presented in Figure 5.7, we did not detect a single mispredicted cost of the optimal kernel partition.

5.5.4 Random sampling approach to a kernel partitioning

In Chapter 4 of the thesis, we present a statistical analysis of the problem of thread assignment for modern multithreaded processors. The results of that analysis demonstrate that a random sample of several thousand random thread assignments likely captures an assignment with a good performance. We analyze the probability that a uniformly distributed random sample of N observations contains at least one observation from the best-performing $P\%$ of the population (e.g. the best 1% of the population). This probability can be computed using the following formula: $Prob = 1 - \left(\frac{\text{100-P}}{100}\right)^{\text{N}}$. As P is a small positive number, the value of the fraction $\frac{\text{100-P}}{100}$ is always between 0 and 1. Therefore, for large N, the factor $\left(\frac{\text{100-P}}{100}\right)^{\text{N}}$ converges to 0, and the observed probability converges to 1. For example, the probability that a uniformly distributed random sample of 1,000 observations contains at least one element from the best 1% of the population exceeds 99.99%.

In order to analyze whether random sampling can be used to select a good kernel partition, we compare the cost provided by the fairly-complex heuristics-based kernel partitioning algorithm proposed by Carpenter et al. [32] with the minimal cost observed in the random sample. The sample was comprised of 20,000 kernel partitions generated using the UD sampling method. The results are shown in Figure 5.10. The X-axis of the figure lists different benchmarks, while the Y-axis shows the possible performance improvement of the kernel partitioning approaches. For *bitonic-sort, des, fft, mpeg2-subset, serpent_full,* and *tde_pp* benchmarks, brute force exploration is feasible, so we compare the kernel partition costs provided by the random sampling and heuristics-based algorithm with the actual minimal costs. For the remaining five benchmarks, *channelvocoder, filterbank, fm, radar,* and *vocoder*, the results are plotted relative to the estimated minimal costs. The minimal cost is estimated using the POT method on 20,000 uniformly distributed random kernel partitions. The markers correspond to the point estimation of the minimal cost, while the error bars correspond to the estimated confidence bounds for

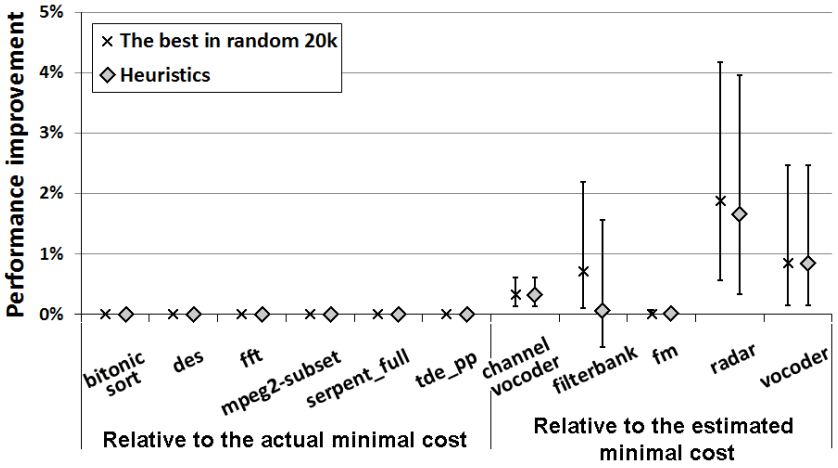


Figure 5.10: Comparison of random sampling (UD method) and heuristics-based algorithm

a 0.95 confidence level.

For *bitonic-sort, des, fft, mpeg2-subset, serpent_full,* and *tde_pp* benchmarks, the best costs observed by the random sampling and the heuristics-based algorithm match the actual best costs of kernel partitions. For the *channelvocoder, filterbank, fm, radar,* and *vocoder* benchmarks, random sampling and heuristics-based algorithm, detect kernel partitions with a cost that is close to the estimated optimal one. For four out of five benchmarks (all except *radar*), the possible improvement of both approaches is below 3% (confidence bounds included). For *radar* benchmark, the estimated improvement ranges up to 4% and 4.2% for random sampling and heuristics-based algorithm, respectively. If we consider the point estimation, the estimated performance improvement is below 2% for all the benchmarks, and below 1% for four out of five benchmarks. For *channelvocoder, fm,* and *vocoder* benchmarks, the performance of the best kernel partitions in the random sample match the performance of the heuristics-based algorithm. For *filterbank* and *radar* benchmarks, the heuristics-based algorithm selected kernel partitions with 0.6% and 0.2% lower cost, respectively, which is a negligible difference. If a good heuristics-based approach is available for the applications, hardware, and metric under study, the user can choose whether to use the heuristics or the random sampling. However, it is common that heuristics-based approaches are not directly applicable to the exact situation under study. It is often difficult and time-consuming to adapt a heuristic to a particular target scenario. On the other hand, the random sampling approach is simple and easy to apply.

Required number of random kernel partitions: The formula $Prob = 1 - \left(\frac{100\text{-}P}{100}\right)^N$

can be used to compute the probability that an uniformly distributed random sample of N observations captures at least one out of $P\%$ of the kernel partitions with the lowest cost. However, as we do not know the difference in the cost in the best $P\%$ of all kernel partitions, the formula cannot be used to compute the difference between the minimal cost captured in a random sample and the actual minimal cost.

In order to analyze whether a sample of N randomly selected kernel partitions captures a good partition, we observe the minimal cost in the random sample and compare it with the minimal cost determined by the statistical estimation or brute force exploration, when feasible. We repeat the experiment for different sample size (from 10 to 20,000 kernel partitions) in order to determine a sample size that is likely to provide a good performance. The random samples are generated with the UD sampling method. Figure 5.11 shows the results of the experiments for *serpent_full* benchmark. The X-axis of the figure lists the number of random kernel partitions in the sample. Dashed vertical lines separate the results for tens (from 10 to 90), hundreds (from 100 to 900), and thousands (from 1,000 to 20,000) of random kernel partitions. The Y-axis shows the relative difference between the minimal cost captured in the random sample and the actual minimal cost determined by brute force exploration. We want to analyze whether a sample of N randomly selected kernel partitions captures a good partition *in general*, i.e. what is *average* performance loss of random sampling approach. In order to present statistically significant results, for each sample size N, we repeat the experiment 100 times[2] and report the mean (cross marker) and the standard deviation (error bars) of performance detected in different runs.

We see that tens of random kernel partitions in the sample are unlikely to capture a good partition. We also detect a high standard deviation, which means that the cost of the best-captured kernel partition is significantly different for different samples of the same size. For hundreds of kernel partitions in the sample, the best captured cost slowly converges to the estimated optimal one. The standard deviation decreases, which means that different samples of the same size provide similar performance. Finally, several thousand random kernel partitions capture a cost that is very close to the optimal one. Also, the detected standard deviation is low (1-2%).

We detect the same trend for all eleven benchmarks under study. Therefore, we conclude that uniformly distributed random sampling can be used to find a good kernel partition. However, we recommend this method only when the random sample contains at

[2]Repeating experiments 100 times is sufficient for statistical analysis of *average* behavior.

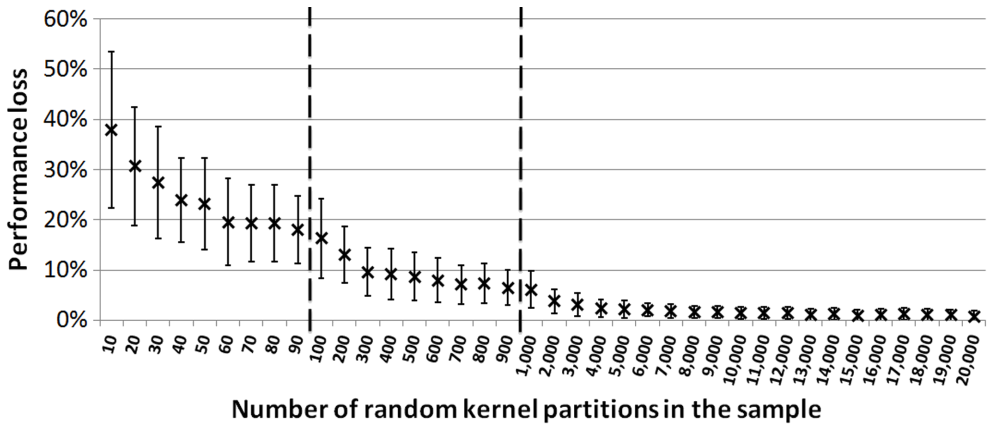


Figure 5.11: The impact of the sample size on the performance of the random sampling (serpent_full benchmark, UD sampling method)

least several thousand kernel partitions.

In order to determine the sample size that captures a good kernel partition, a user could also observe the convergence rate of the best observed kernel partition performance in the sample as the sample increases. For example, from the Figure 5.11, we could observe that increasing the random sample over several thousand kernel partitions insignificantly improves the observed performance, and stop the random sampling at that point without estimation of the optimal performance. Although this approach may provide good results, it is not clear how it would avoid convergence to a local minimum of the population (e.g. see the results for 60, 70, and 80 kernel partitions in Figure 5.11). Also, without estimated optimal performance, we cannot determine the quality of the delivered kernel partition, i.e. we cannot provide the confidence bounds of the estimated performance improvement.

5.5.5 Other considerations

There are several additional aspects to consider regarding the presented statistical approach.

Experimentation time: The presented approach requires thousands of random kernel partitions to be generated and evaluated. The time to generate and evaluate 1,000 uniformly distributed random kernel partitions, for one of the StreamIt 2.1.1 benchmarks, was on average 28 minutes, using a single core of an Intel Xeon E5649 processor at 2.5GHz with 4GB memory. Running the POT method takes less than a minute. The program that generates the uniformly distributed samples is not optimized and is imple-

mented in the Python programming language. An implementation in C would be much faster. Also, since different kernel partitions can be generated independently, the time to generate the sample would decrease linearly with the number of cores. The experimentation time is acceptable considering that the selected kernel partition will be compiled to the executable that can be used on numerous systems based on the same hardware during the lifetime of the system.

Scalability: The number of cores and the number of hardware threads increase in each processor generation [116]. In order to optimally use future multicore processors, kernel partitioning algorithms will have to generate a significantly larger number of threads. It is important, therefore, to analyze how these algorithms scale with the number of software threads. On the other hand, at the application level, it is reasonable to expect that the complexity of streaming applications increases leading to more complex stream graphs that comprise a larger number of kernels.

The statistical analysis that estimates the optimal kernel cost is based on the values of the performance metric, so its cost is independent of the number of threads and the complexity of the stream graph. The cost of the sampling method that generates the uniformly distributed random kernel partitions scales linearly with the number of kernels in the stream graph and with the number of output software threads. It also scales linearly with the mixing time of the partition graph [102].

Compiler optimizations and system constraints: When the program is described using a stream language, the compiler may perform complex optimizations over the stream graph; it can combine adjacent filters, split computationally intensive filters into multiple parts, or duplicate filters to have more parallel computation [2]. In order to find the set of optimizations that provides the best performance, it is important to determine good kernel partitions for different optimization sets. In this case, the proposed kernel partitioning approach does not change, but the random sampling and the presented statistical analysis are simply repeated for different optimization sets, i.e. for different stream graphs of the same program. Kernel partitioning that satisfies different system constraints, such as optimizing performance subject to memory limits, is an interesting avenue for future work.

Evaluation on real hardware: The presented statistical approach was evaluated based on the estimates of the kernel partitions' costs provided by the StreamIt compiler. The approach was not evaluated on real hardware because of limitations in the experimental environment. The back-end of the StreamIt compiler, that we used in the study,

was not capable of generating working code for different user-defined kernel partitions. As a part of future work, we plan to evaluate the presented statistical approach on real hardware. In order to do so, we intend to modify the StreamIt compiler, so it can generate the executables that correspond to any given kernel partition.

5.6 Related Work

Several projects and studies propose different tools for compiling of streaming-like applications and their mapping onto multicore architectures.

StreamIt is a project with publicly available compiler and benchmark suite [3] . The StreamIt source language imposes a structure on the stream program graph to the compiler. The StreamIt compiler performs fully automated load balancing, communication scheduling, routing, and a set of cache optimizations [72, 73, 138]. StreamIt compiler targets the Raw Microprocessor [157], symmetric multicore architectures, and clusters of workstations.

The Stream Graph Modulo Scheduling (SGMS) algorithm is part of StreamRoller [97], a StreamIt compiler for the Cell Architecture. This algorithm splits stateless kernels, partitions the graph, and statically schedules the software threads onto the Cell architecture. The splitting and partitioning problem is translated into an integer linear programming problem, which is solved using CPLEX Optimization Studio [83], an software package for mathematical programming.

Gedae Graph Language [105] is a proprietary GUI tool that supports the hierarchical development of data flow graphs. Gedae allows the user to specify different graph partitions and automatically maintains the data flow and connectivity of the graph. However, all the graph partition is done under user control, not by the compiler.

The Ptolemy II software environment [58] is designed to model heterogeneous embedded computing systems. Ptolemy views computing systems as a set of basic processing blocks (*actors*) that are connected using explicitely-defined communication channels. This view is very similar to the state-of-the-art interpretation of streaming-like applications. Related work from the Ptolemy project explores the more theoretical aspects of partitioning and scheduling data flow graphs for multiprocessors [77].

Liao et al. [103] present a parallel compiler for the Brook streaming language [30] with aggressive data and computation transformations. The compiler models each streaming kernel as an implicit loop nest over stream elements and uses affine partitioning to map

regular programs onto multicore processors.

Farhad et al. [63] show that state-of-the-art linear programming approaches are impractical for transformations of large stream graphs to be executed on a large number of processor cores. The authors also propose an approximation algorithm for deploying stream graphs on multicore processors.

Our study shows a different approach to kernel partitioning problem. Instead of using complex heuristics-based algorithms, we address the problem using random sampling and statistical inference. We present a statistical method that estimates the performance of the optimal kernel partition based on measured performance of a sample of random partitions. We also demonstrate that random sampling can be used to find a kernel partition with performance close to the optimal one.

Our study improves the compilation of the streaming application source code into multithreaded executable. In Chapter 4 we used random sampling and statistical inference to analyze the optimal assignment of existing software threads onto different processor cores. These two studies address fundamentally different problems that complement each other. In its essence, kernel partitioning is a graph partitioning problem, and the thread assignment problem addressed in Chapter 4 is a multiprocessor scheduling problem [67].

5.7 Summary

One of the greatest difficulties in using modern computing systems is how to write efficient, portable, correct software for multicore processors. A promising approach is to expose more parallelism to the compiler through domain-specific languages, enabling the compiler to perform complex high-level transformations. One important application domain comprises stream programs. An important step in compiling a stream program to multiple processors is kernel partitioning, which significantly affects application performance. Finding an optimal kernel partition is, however, an intractable problem.

In this chapter of the thesis, we proposed a statistical approach to the kernel partitioning problem. We described a method that statistically estimates, with a given confidence level, the performance of the optimal kernel partition. Knowing the optimal performance improves the evaluation of any kernel partitioning algorithm, and it is the most important piece of information for the system designer when deciding whether an existing algorithm should be enhanced. We demonstrated that the sampling method is an important part of the analysis, and that not all methods that generate $i.i.d.$ samples provide good results. We

also showed that random sampling on its own can be used to find a good kernel partition, and that it could be an alternative to heuristics-based approaches.

The presented statistical method does not depend on the application. It does not require any application profiling nor does it require the understanding of the application stream graph. The method can be applied to streaming applications with any number of kernels, and it can target any number of software threads. The presented method can analyze different metrics such as throughput, maximum hardware utilization, and minimum energy or power consumption.

We successfully applied the presented statistical analysis to the benchmarks included in the StreamIt 2.1.1 suite. The method precisely estimated the optimal kernel partition performance for all the benchmarks under study. Also, in our experiments, several hundred or several thousand random kernel partitions were enough to find a partition with close to optimal performance. The performance of the kernel partitions that were selected using random sampling matched the performance provided by the complex heuristic-based approach.

Chapter 6

Evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments

Commercial Off-The-Shelf (COTS) processors are now commonly used in real-time embedded systems. The characteristics of these processors fulfill system requirements in terms of time-to-market, low cost, and high performance-per-watt ratio. However, multithreaded processors are still not widely used in real-time systems because the timing analysis is too complex. In multithreaded processors, simultaneously-running tasks share and compete for processor resources, so the timing analysis has to estimate the possible impact that the inter-task interferences have on the execution time of the applications.

In this chapter, we propose a method that quantifies the slowdown that simultaneously-running tasks may experience due to collision in shared processor resources. To that end, we designed benchmarks that stress specific processor resources and we used them to: (1) Estimate the upper limit of a slowdown that simultaneously-running tasks may experience because of collision in different shared processor resources, and (2) Quantify the sensitivity of time-critical applications to collision in these resources. We used the presented method to determine if a given multithreaded processor is a good candidate for systems with timing requirements. We also present a case study in which the method is used to analyze three multithreaded architectures exhibiting different configurations of resource sharing. Finally, we show that measuring the slowdown that real applications experience when simultaneously-running with resource-stressing benchmarks is an important step in measurement-based timing analysis. This information is a base for incremental verification of multithreaded COTS architectures.

6.1 Introduction

Commercial Off-The-Shelf (COTS) processors are increasingly being considered in the design of real-time and mission-critical embedded systems in order to reduce the non-recurring engineering and time-to-market costs [24]. In such systems, ensuring timing predictability and meeting deadlines are of prime importance and therefore the analysis of the system and the target applications are essential before deployment [160]. Time predictability is, in fact, a requirement not only in the real-time market, but also coming to be of primary importance in the mainstream market as recognized in the HiPEAC roadmap [55].

Currently, the COTS processor market is moving toward multithreaded (MT) [1] processor architectures. Multithreaded COTS processors are of special interest due to their good performance-per-watt ratio and high performance opportunities [152]. These architectures are particularly well suited for embedded *integrated architectures* in which several functions are integrated into the same processor, such as Integrated Modular Avionics (IMA) [158] in the avionics domain or Automotive Open System Architecture (AUTOSAR) [22, 113]. In this context, multithreaded processors can potentially schedule mixed criticality workloads, i.e. workloads composed of safety-critical, mission-critical, and non-critical applications inside the same processor, improving the hardware utilization and so reducing cost, size, weight, and energy consumption [110].

Unfortunately, despite the benefits that MT COTS processors may offer in embedded real-time systems, particularly in integrated architectures, the time-critical market has not yet embraced such a shift. The main challenge that MT COTS architectures face is with predicting the impact that the collision among simultaneously-running tasks has on execution time of time-critical tasks. The loss of predictability is explained by the fact that co-running tasks have to share the hardware resources, even if they do not communicate with each other. When two or more tasks that share a hardware resource try to access it at the same time, the tasks experience *inter-task interference*. Inter-task interferences are handled by an arbitration mechanism, which may affect the execution time of running tasks. As a result, it is much more difficult to provide the worst-case execution time (WCET) estimations for applications running on MT processors than running on single-threaded

[1] We will use the term "multithreaded processor" to refer to any processor that has support for more than one thread running at a time. Multicore, HyperThreading, Simultaneous Multithreading, Coarse-grain Multithreading, Fine-Grain Multithreading processors, or any combination of them are multithreaded processors.

processors. Several studies show that collision in processor resources between co-running tasks may cause significant impact to application execution time [54, 154] and therefore on WCET [120].

Static WCET analysis computes WCET bound based on the extensive program analysis and detailed model of the hardware [126]. Static WCET analysis is currently the only approach that computes safe WCET bounds, i.e. that guarantees that the actual execution time of the program cannot be longer than the computed WCET bound. However, there are several reasons that make the use of static WCET analysis difficult on real industrial programs running on MT COTS architectures [92, 94, 111]. Some of these reasons are: (1) Static WCET analysis of real industrial programs with a vast number of possible execution paths is a challenging task. (2) The implementation of accurate hardware models for new architectures requires a significant effort and a detailed description of the hardware, which is not always available. (3) Possible interference in shared hardware resources among tasks that simultaneously execute on MT architectures significantly increases the complexity of timing analysis.

This has motivated studies that analyze if changes in hardware can facilitate the effective timing analysis of real industrial programs running on MT architectures. There have been several hardware proposals [18, 68, 78, 110, 124, 125] to ease the computation of composable WCET bounds or WCET estimates [2] of tasks running on multithreaded architectures. However, these proposals require changes in hardware and additional features that the current MT architectures do not have. Therefore, the industry that wants to use MT architectures in their current real-time system designs cannot benefit from them.

A measurement-based approach for single-threaded architectures computes the *WCET estimate* by multiplying the longest observed execution time (LOET) by a *safety margin*, usually provided by an expert with understanding of the target hardware architecture and the reference applications (see Figure 6.1) [111]. This method has been successfully used in the past to determine WCET estimate of applications running on single-threaded processors with a moderate difference between the Average Case Execution Time (ACET), the LOET, and the WCET estimate. However, the method provides no analytical guarantees that the estimated WCET is safe and it may depend significantly on the quality

[2]The term *WCET bound* is used to refer to *safe upper bound* of program execution time that is provided by static WCET analysis. Static WCET analysis guarantees that the execution time of a given program will not exceed its *WCET bound* for all valid input configurations. *WCET estimate* refers to *the upper bound of program execution time that is unlikely to be exceeded*. The techniques that compute *WCET estimate* do not provide formal verification that program execution time does not exceed *WCET estimate* [92, 94].

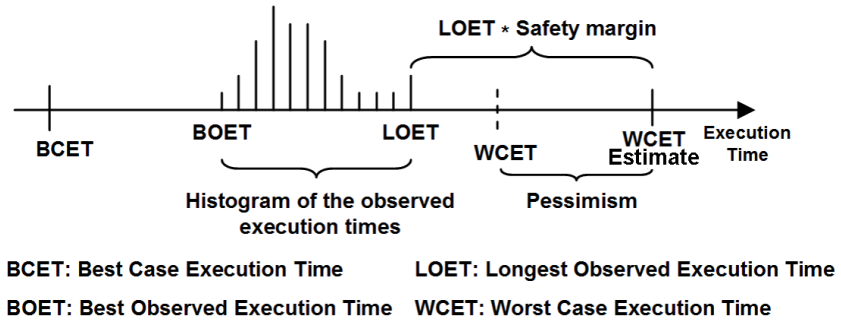


Figure 6.1: Measurement-based timing analysis

of the test-cases used as well as the experience of the expert(s) who compute the safety margin. A direct extension to the measurement-based analysis presented in Figure 6.1 to MT COTS processors would consist of running several reference applications simultaneously on the same processor, and monitoring the execution time for each application in the workload. However, in the case of MT architectures, the design of test-cases and the choice of the workload has more importance than for single-threaded processors. The effect of sharing processor hardware resources with simultaneously-running (*co-running*) tasks may introduce significant variations in the execution time of applications. This can make a methodology based on safety margins fail. In addition to this, a change of any co-running task may affect the way that co-runners interfere, and necessitates repeating the WCET analysis for all the tasks in the workload. Thus, the measurement-based timing analysis used for single-threaded processors cannot be directly extended for MT COTS architectures.

Our study provides to the industry a systematic methodology for measurement-based timing analysis of applications running on MT COTS architectures. The main contributions of our study are:

- We show that running workloads composed of real applications may not be sufficient to determine the slowdown that simultaneously-running tasks may experience because of collision in shared processor resources. Thus, the measurement-based timing analysis used for ST processors cannot be directly extended to MT architectures.

- We present a method to determine if a given MT processor is a good candidate for systems with timing requirements.

- We also show that measuring the slowdown that real applications experience when co-running with designed resource-stressing benchmarks improves measurement-based timing analysis for MT architectures. This can be used as a base for incremental verification, a key feature of integrated implementations such as IMA or AUTOSAR.

In order to reach these objectives, we defined a set of specific *resource-stressing* benchmarks that introduce a high number of interferences on each potentially-shared hardware resource. By using these resource-stressing benchmarks as co-runners, we obtain a good estimation of the worst-case slowdown that real applications may experience because of collision in shared processor resources. When a workload is composed only of resource-stressing benchmarks, the detected slowdown is unlikely to be exceeded for any workload composed of real applications. Therefore, the slowdown detected when using resource-stressing benchmarks may serve as an upper estimate of the effect of inter-task interference for a given processor.

We present several case studies in which we analyze three MT COTS architectures with different degrees of shared resources. We show that, for a given workload composed of several benchmarks, all three architecture types show low interference among co-running tasks and stable execution times. However, our method shows that the potential variation in the execution time of applications is different for each architecture under study.

As our study targets real COTS processors, we do not suggest any hardware change in the target architecture, but propose a way to improve the measurement-based approach for MT COTS processors. This is one of the main differences with previous works in the field that have suggested hardware modifications to improve architecture time predictability.

The rest of the chapter is organized as follows: In Section 6.2, we propose a method for analysis of potential interference of co-running tasks in shared processor resources. Section 6.3 shows details of the experimental environment and methodology used in the study. Section 6.4 presents a case study in which we evaluate the suitability of three MT COTS architectures for time-critical environments. Related work is presented in Section 6.5, while Section 6.6 summarizes the study.

6.2 Analysis of Inter-Task Interferences in current MT processors

In MT processors, the execution time of a task depends not only on the underlying hardware and the way the task is programmed, but also on the slowdown caused by the interference between simultaneously-running tasks. In order to provide a meaningful WCET estimation for tasks running on an MT processor, it is required to take into account the inter-task interference in shared processor resources. Several studies [49, 118, 119] and projects [18, 68, 78, 110, 124, 125, 150] address this problem for safety-critical applications by introducing hardware mechanisms to define the upper bound of the delay a task can experience due to interferences in cache memory, bus, and access to the main memory. Unfortunately, these proposals have not yet been accepted by processor manufactures, and it is unclear if the COTS processor market will adopt them. Even if the hardware proposals were accepted, it would take at least five to ten years to implement them.

The analysis of the impact of inter-task interferences to application WCET is very complex. Without the hardware support proposed in the previously mentioned studies and projects, using static WCET analysis for MT COTS processor running real workloads is infeasible in practice. As we mentioned in Section 6.1, another solution could be to directly extend the measurement-based approach used for single-threaded architectures, and to estimate the application's WCET based on the longest observed execution time within *all possible* workloads.

The problem with this approach is that the number of different workloads in real systems may be large. For example, assume a task set composed of n tasks is to be executed on a target processor able to run up to k tasks at a time, in which $n > k$. Under this scenario, the number of workloads that have to be analyzed is $\frac{n!}{k!(n-k)!}$. Moreover, any change in the workload, e.g. a shift in time at which each application in the workload starts, would invalidate the previous analysis for all running tasks, especially when running mixed-criticality workloads with non real-time applications. Hence, measurement-based timing analysis considering all possible workloads is not feasible in practice for time-critical tasks running on MT COTS processors.

We propose a software solution to this problem. In particular, we propose the design of *resource-stressing* benchmarks and a methodology for their use to quantify possible slow-

down that co-running tasks may experience because of a collision in shared resources of MT COTS processors. The objective of the resource-stressing benchmarks is to introduce a high-load onto each of the processor hardware resources that the task which is under the analysis may use. Thus, the resource-stressing benchmarks can be used to provide good estimates of the potential slowdown that a set of simultaneously-running real applications may experience because of the collision in shared resources of a given processor. When the methodology is used on different MT COTS architectures, it can help to determine the suitability of each architecture to time-critical environments. The methodology can also be used for a given architecture to determine the potential variation in execution time that a particular task in a workload may experience if any of the co-runners change. This information is a key to providing incremental qualification [60].

6.2.1 Worst-interference benchmark

When designing a resource-stressing benchmark, we would like to have the *worst-possible interference benchmark*. This benchmark would cause the highest possible interference in the shared resources that are used by the application under study. In this scenario, the slowdown of the application due to collision in processor resources could be used to compute a safe execution time bound. It is important to note that the worst-interference benchmark would be specific for the application under study and the target MT processor.

Unfortunately, the design of the worst-interference benchmark is extremely difficult or even impossible, even for a single hardware shared resource. In order to illustrate this, we analyze the design of a benchmark that should cause the worst interference to a given application in the shared instruction fetch unit (IFU) in an SMT processor. In this case, we assume that the IFU is designed to fetch up to one instruction in each cycle, and implements a least recently fetched policy, i.e. the thread that last fetched an instruction has the lowest priority. Under this fetch policy, whenever the worst-interference benchmark fetches an instruction, the application under study becomes the least recently fetched task with higher priority in the next access to the IFU. In order to design the worst-interference IFU benchmark for a given application, it is required to: (1) Determine precise cycles in which application under study will access IFU, (2) Determine the priority of the application in each IFU access, and (3) Consider how the interference between the application under study and the worst-interference benchmark and will delay any future IFU requests of both tasks.

In general, in real time systems, the design of worst-interference benchmark requires:

- Full knowledge of the processor implementation, including resource latency, arbitration policy, etc. However, many of these hardware features of COTS processors are not reported in the public documentation.

- Full knowledge of the application, including input data set, execution flow, resource usage pattern, etc. However, a common practice is that applications are provided by different suppliers (Tier 1) [69], making it difficult to have full knowledge about all applications.

- The worst-interference benchmark should stress all resources used by the application under study, needing to be perfectly aligned in the access to each of shared resources used by the application. Any misalignment between worst-interference benchmark and application under study can significantly reduce the impact of resource sharing.

All these requirements make the design of worst-interference benchmarks infeasible in practice.

6.2.2 Resource-stressing benchmarks

A key design choice in our resource-stressing benchmarks is how to stress shared processor resources. (1) We can design benchmarks that specifically stress a single resource by putting a high load on it. For example, to stress the instruction fetch unit, we could design a benchmark that fetches an instruction in each cycle. The downside of this solution is that only one resource can be stressed at a time. (2) Alternatively, we can design benchmarks that stress several resources at a time. For example, a benchmark can comprise different instructions that access different resources. The downside of this solution is that the stress in each resource decreases as several resources are stressed simultaneously.

In architectures with two cores or hardware contexts, in which we can only run the application under study and one resource-stressing benchmark at a time, it is unclear which of the two approaches better show the sensitivity of applications to hardware resource sharing. However, as the number of cores or contexts increases, the methodology that uses benchmarks that stress a single hardware resource scales very well. Under that methodology, we can reserve 1 out of the N cores (or hardware contexts) to run the application

under study on, and use the remaining N-1 to run different combinations of stressing benchmarks, each stressing a given resource. This improves the obtained slowdown as with several cores or hardware contexts we can observe the execution of the application under different resource-stressing conditions. In Section 6.4.4, we show a case study in which we use this feature of the methodology.

We start benchmark design by identifying common levels in the hardware resources shared in MT COTS processors. This allows the creation of a set of benchmarks that stress those particular resources. In particular, we identify three *resource sharing levels*:

(1) Intra-core resources include: (1.1) Front end of the pipeline; (1.2) Back end of the pipeline: Integer and Floating Point (FP) execution units; (1.3) The L1 data cache; (1.4) The L1 instruction cache.

(2) Inter-core resources such as the L2 cache memory (the last level of cache memory).

(3) Interface to off-chip resources such as the bandwidth to the main memory.

Although these levels of shared resources are common in MT COTS processors, the effect in time (delay) that the interference in each of these resources causes, depends on the particular processor. To measure that delay, for each level of shared resources, we designed at least one resource-stressing benchmark as described next:

(1.1) Front end of the pipeline: In order to stress front end of the pipeline, mainly the instruction fetch unit and the decode unit, we designed a benchmark that executes a series of *nop* instructions. The nop instruction is a low latency instruction that puts significant stress to the front end of the pipeline and negligibly stresses rest of the processor resources.

(1.2) Back end of the pipeline (Integer and FP execution units): In state-of-the-art processors, different integer and FP instruction may execute in different execution units and may have different behavior (e.g. may be pipelined or non-pipelined). To cover the different cases of possible interference, we design six benchmarks for stressing integer and FP execution units. *intAdd*, *intMul*, and *intDiv* benchmarks consist of a sequence of integer addition, multiplication, and division instruction, respectively. *fpAdd*, *fpMul*, and *fpDiv* benchmarks execute a serial of floating point addition, multiplication, and division instructions, respectively.

(1.3) The L1_dcache benchmark consists of a sequence of load instructions that access different cache lines of the L1 data cache. The size of the array that the benchmark

traverses is the same as the size of the L1 data cache. Therefore, when the L1_dcache benchmark executes in isolation, most of the loads hit in the L1 data cache. However, when L1_dcache cache is scheduled with a co-runner that uses L1 cache, the data sets of both benchmarks do not fit in the cache, which causes L1 misses and longer execution time.

(1.4) The L1_icache benchmark consists of a sequence of jump instructions that access different cache lines in the instruction cache. The size of the code is equal to the size of the instruction cache.

(2) The L2 benchmark is designed using the same principle as L1_dcache benchmark. The only difference is that the size of the array that the benchmark traverses is equal to the L2 cache size.

(3) Interface to off-chip resources (Memory bandwidth): One of the factors which has a significant impact on the applications performance is access to the off-chip resources. The *mem_bw* benchmark is a resource-stressing benchmark that has the same structure as L1_dcache or L2 benchmark (a sequence of load instructions that traverse the array). Since the purpose of the benchmark is to stress the bandwidth to memory, but not to cause any collision in the main memory, the size of the array that the benchmark traverses has to be chosen to cause misses in the last level of cache, but not to cause page faults in the main memory. For the configurations we test in this study, the size of the *mem_bw* array is four times larger than the size of the last level of cache.

The set of resource-stressing benchmarks can be easily extended to stress execution units and I/O interfaces of different processors. For each architecture under study, the user should determine the set of shared resources in which co-running tasks may collide and design resource-stressing benchmarks for each of them.

6.2.3 Implementation

The framework for automatic execution of the experiments and for processing of the results is implemented in C programming language using POSIX threads [31]. The real-time and resource stressing benchmark that are used in each experiment are read from the input file defined by the user.

The core of our framework consists of benchmarks that stress specific processor resources. Each resource-stressing benchmark is defined in the function that is included in the framework. The benchmark functions are implemented directly in assembly of the

Table 6.1: Structure of $intAdd$ resource-stressing benchmark

Line	Source code	Explanation
001	movl %1, %ecx	initialize loop counter ecx (%1 is an input parameter)
002	label_intAdd:	beginning of the loop
003	add %eax, %ebx	target instruction
004	add %ebx, %eax	target instruction
...	...	...
...	...	...
252	add %ebx, %eax	target instruction
253	decl %ecx	decrement loop counter
254	cmp %ecx, $0	compare loop counter with 0
255	jne label_intAdd	if (counter != 0) jump to the beginning of the loop

target processor in order to: (1) Provide the programmer with the maximum control over the instructions that are to be executed, and (2) Prevent a compiler from applying any optimization that changes the core of the benchmarks. The assembly functions are inlined in a C code in order to avoid the overhead of the function call.

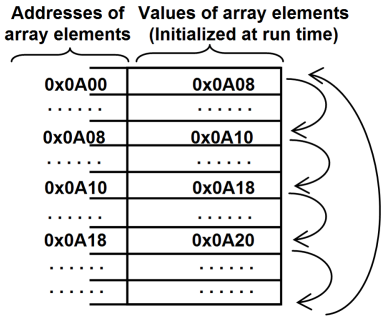
All the resource-stressing benchmarks are designed using the same principle that is presented in Table 6.1. Each benchmark is comprised of three parts: (1) The register used as a loop iteration counter (ecx) is initialized to the value of the input parameter (line 001). The initial value of ecx determines the number of loop iterations and the duration of the benchmark. (2) The main part of the benchmark is a sequence of instructions of target type (lines from 003 to 252). The add instruction is changed by the corresponding target instruction in each of the resource stressing benchmarks: nop instruction [3] in the case of the *nop* benchmark, integer multiplication in the case of the *intMul* benchmark, etc. (3) Finally, the sequence of target instructions is followed with the decrement of the loop counter register (line 253), comparison of the counter value with zero (line 254), and a conditional branch to the beginning of the loop (line 255). The overhead of the loop and the calling code is very low - more than 99% of the instructions targets the specific resource we want to stress.

An overview of the benchmarks that stress the cache memory and the memory band-

[3] In the current version of the nop resource-stressing benchmark, we use one-byte nop instruction which is an alias mnemonic for the *XCHG (E)AX, (E)AX* instruction in Intel architectures. This instruction performs no operation and does not impact machine context, except for the instruction pointer register [1]. The same effect could be achieved by using multi-byte nop in processors that have a support for this instruction [1]. An interesting avenue of future work could be to analyze whether using more complex opcodes would put higher stress to the decode unit and cause higher collision among simultaneously-running tasks that share this processor resource.

```
for(cnt=0; cnt < array_size; cnt+=stride)
{
    if(cnt<array_size-stride)
    {
        // Each array element contains the address of ...
        // ... the following array element we want to access.
        array[cnt] = (int)&array[cnt+stride];
    }
    else
    {
        // The last accessed element in the array points ...
        // ... to the first element of the array that we access.
        array[cnt] = (int)array;
    }
}
```

(a) Initialization code



(b) An example of a memory access pattern

Line	Source code
001	movl %1 %ecx
002	movl %2, %eax
003	label_mem:
004	mov (%eax), %eax
005	mov (%eax), %eax
...	...
...	...
253	mov (%eax), %eax
254	decl %ecx
255	cmp %ecx, $0
256	jne label_mem

(c) Assembly stressing code

Figure 6.2: Overview of the memory stressing benchmarks

width is shown in Figure 6.2.3. The benchmarks are implemented using the concept of *pointer chasing*. In the benchmark initialization, which is done in the C code, we allocate a contiguous section of memory and initialize it in such a way that a given array element contains the address of the next array element (memory location) that we want to access, see Figure 6.2.3(a). The benchmarks are initialized to (1) Traverse the whole array, and (2) Access different cache lines in each memory access. An example of a benchmark memory access pattern is shown in Figure 6.2.3(b). Finally, Figure 6.2.3(c)

shows the assembly code that stresses the memory subsystem. First, the register used as a loop iteration counter (ecx) is initialized to the value of the input parameter (line 001). The initial address of the array is passed to the assembly code as an input parameter (line 002). The main part of the benchmark (lines from 004 to 253) is a sequence of indirect load instructions ($mov(\%eax), \%eax$) that follow the memory access pattern specified in the initialization.

In order to design the L1_icache benchmark, we used a sequence of unconditional jump instructions that point to different labels (jump destinations). The code is designed in such a way that, at run-time, the benchmark (1) Traverses the whole instruction cache, and (2) Accesses different cache lines in each jump instruction.

We analyzed not only the functionality of the resource-stressing benchmarks, but also their portability to different processors with different cache organization or different ISA. By setting the values of parameters $array\_size$ and $stride$ to proper values (see Figure 6.2.3(a)), a user can easily adjust the memory stressing benchmarks to stress different parts of the memory subsystem of different architectures. The stressing assembly code of the L1_icache benchmark is automatically generated. In order to adjust the L1_icache benchmark to stress instruction caches with different size and organization, the user only has to specify the desired size of the stressing code and the stride between consecutive jump instructions (i.e. the stride between consecutive accesses to instruction cache). The resource stressing benchmarks are implemented in x86 ISA. In order to port benchmarks to architectures with different ISA, e.g. POWER or SPARC, the programmer only needs to change the x86 instructions in the assembly stressing code with the corresponding instructions of the target ISA. The assembly stressing code of the benchmarks is very simple, see Table 6.1 and Figure 6.2.3(c), thus a little effort is required to port this code to architectures with different ISAs.

6.2.4 Using the resource-stressing benchmarks

We used the resource-stressing benchmarks to: (1) Estimate the upper limit of a slowdown that co-running tasks may experience because of collision in different shared resources of a processor. This slowdown can be used to determine if a given MT processor is suitable for a time-critical environment. (2) Quantify the possible impact of inter-task interference to the execution time of an application. This improves the WCET estimation

for applications that execute in MT COTS processors.

6.2.4.1 Worst-case slowdown in shared processor resources

In order to quantify the slowdown that an application may experience due to collision in the different shared resources of a given MT COTS processor, we deploy our resource-stressing benchmark in the following way. First, we measure the execution time of each resource-stressing benchmark when it runs in isolation ($ET_{isolation}$). Second, we measure the execution time when several resource-stressing benchmarks run concurrently (ET_{MT}). In order to quantify the interference in processor resources, we compute the slowdown or normalized execution time as the relative difference between the benchmark execution time when it shares processor resources with other co-running benchmarks and when it runs in isolation: $slowdown = \frac{ET_{MT}}{ET_{isolation}}$. As resource-stressing benchmarks put high load on different hardware resources of a processor, the computed slowdown presents a good estimation of the worst-case slowdown that real applications may experience because of collision in these resources. Understanding the slowdown that co-runners may experience because of collision in shared resources can be used to define the suitability of a processor for a time-critical environment.

- When co-runners experience a significant slowdown due to resource sharing, this shows a potentially high variation in execution time of applications running on the processor. This means that the processor is not a good candidate for systems that have timing requirements.

- When co-runners experience a low slowdown due to resource sharing, it means that the processor is suitable for running time-critical applications. Low variation in execution time would allow accurate timing analysis for applications running on these architectures even if the set of co-runners change.

6.2.4.2 WCET estimation for real applications

In order to provide a meaningful WCET estimate of real applications running on a given MT COTS processor, it is important to quantify the slowdown that applications may experience due to collision in shared processor resources. Resource-stressing benchmarks can be used to measure this slowdown. In order to quantify a slowdown due to inter-task interferences in shared processor resources, we deploy our resource-stressing benchmark in

the following way. We measure the execution time of a real application when it runs in isolation ($ET_{isolation}$) and when it simultaneously executes with different resource-stressing benchmarks ($ET_{RS[i]}$). The *sensitivity* of the application to sharing *a given resource* X of the processor is computed as $sensitivity[X] = \frac{ET_{RS[X]}}{ET_{isolation}}$, where $ET_{RS[X]}$ is the execution time of the application when co-running with a resource-stressing benchmark that targets resource X.

The *sensitivity* of an application running on a given processor is computed as the maximum value of sensitivity for all analyzed processor resources. If this sensitivity low, the application is not sensitive to the resource sharing on a given processor. This may be because the application does not stress shared processor resources. If the application experiences a significant slowdown when it executes with a benchmark stressing a given processor resource, the application will show a high sensitivity to that resource. Collision in that hardware resource is a potential source of high execution time variation that the application may experience.

In processors with more than two virtual CPUs (i.e. cores or hardware contexts), we can run the real application under study and several resource-stressing benchmarks at a time. These experiments combine the effect of the interferences in different resources, which may increase a measured slowdown and improve the WCET estimation. We expect that this methodology will show even better results in future MT COTS processors in which the number of cores and hardware contexts will increase.

6.3 Experimental Environment

In this section, we present the experimental environment used in the case study. We describe the different MT COTS processors, real benchmarks, and the methodology used in the experiments.

6.3.1 Hardware environment

We present a methodology that can be used to analyze how collision in shared processor resources can impact on the execution time of an application running on MT architectures. Still, in order to determine if a given processor is suitable for embedded real-time systems, it is important to consider several characteristics such as performance, energy consumption, dissipation, etc. The processors we used in the case studies are not neces-

sarily the type of MT COTS processors used in embedded real-time systems because they do not necessarily meet those requirements. However, the processors used exhibit several configurations of resource sharing that allow us to show the application of our methodology for different types of target architectures. The three MT COTS processors considered in this study are the following:

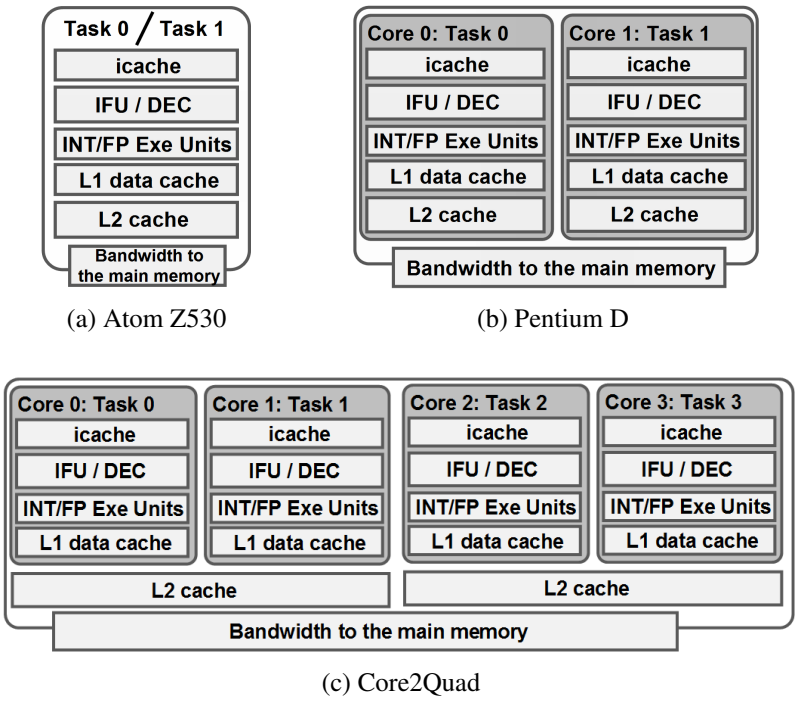
(1) Atom Z530 [21] is a HyperThreading processor. The schematic view of the processor is shown on in Figure 6.3(a). Atom Z530 has one core that supports the simultaneous execution of two tasks (two software threads). Most of the processor resources are shared among co-running tasks: from the front-end of the pipeline to the memory bandwidth. The platform based on the Atom processor contains 500MB of main memory.

(2) The **Pentium D** processor [121] contains two cores, and each of them can execute only one task at a time, see Figure 6.3(b). The front end of the pipeline, integer and FP execution units, L1 data, instruction, and L2 caches are private to each core. Simultaneously-running tasks on this processor can only collide in the bandwidth to the main memory. Platform based on Pentium D processor contains 2GB of main memory.

(3) The **Core2Quad** processor [46] that we use in the study (Q9550) contains four cores, and each of them can execute only one task at a time, see Figure 6.3(c). The front end of the pipeline, integer and FP execution units, L1 data and instruction caches, are private to each core. The L2 cache memory has two partitions. Each partition is shared by tasks running on two cores: Partition 1 is shared by tasks running on Core 0 and Core 1, while partition 2 is shared among Core 2 and Core 3. This means that the distribution of simultaneously-running tasks on the processor determines if tasks share the L2 cache.

In order to evaluate all the possible scenarios, we run two sets of experiments for the Core2Quad processor: (1) Reference and stressing benchmarks are bound to Core 0 and Core 1, while no benchmarks are running on cores 2 and 3 (RS-nn). In this distribution, reference and stressing benchmarks share the L2 cache. (2) A reference benchmark is bound to Core 0, while no benchmark is running on Core 1, and two stressing benchmarks are running on Core 2 and Core 3 (Rn-SS distribution). In this distribution, reference and stressing benchmarks do not share L2 cache. Finally, on the Core2Quad processor, the bandwidth of the main memory is shared among all simultaneously-running tasks. The platform based on the Core2Quad processor contains 4GB of main memory.

As mentioned in Section 6.2, when a processor contains more than two cores of hardware contexts, we can run several combinations of resource-stressing benchmarks in or-



(a) Atom Z530

(b) Pentium D

(c) Core2Quad

Figure 6.3: Schematic view of the processors used in the study

der to improve the worst-case slowdown obtained by our methodology. In the case of the Core2Quad processor, we will bind the benchmark under study to Core 0, and run all possible combinations of L2 and mem_bw benchmarks in Core 1, Core 2, and Core 3.

6.3.2 Benchmarks

In addition to the set of resource-stressing benchmarks presented in Section 6.2, we used the following benchmarks to evaluate the proposed methodology.

STAP Radar: Space Time Adaptive Processing (STAP) Radar is a mission-critical application developed by Thales Research & Technology [43]. The application is a simplified view of a moving target indication application, whose goal is to receive the echo of a periodic sequence of radar pulses and to detect the objects that are moving on the ground. The main characteristics of the applications are the following. First, a large part of the application is data-flow, manipulating multidimensional arrays of data. Second, data reordering (switching dimensions of arrays) is often needed. Third, the processing chain uses different operators, with different specific needs in terms of precision and dynamic range. And, fourth, real-time performance is one of the key requirements, both in terms of computation throughput and latency.

CoreMark [47] is a benchmark developed by the Embedded Microprocessor Benchmark Consortium (EEMBC) [57] and is designed specifically to test the functionality of a processor core. The EEMBC CoreMark suite contains three types of functionality that are representative of real-time environments: (1) Matrix-related functionality: Matrix multiplications, matrix addition, addition of a constant to a matrix; (2) Linked list management operations: Operations like insert, remove, or find the element in the linked list; (3) Finite State Machine (automata) operations. In each run, CoreMark benchmark executes all three sets of functions.

H264_encode is a benchmark included in MediaBench II suite [66]. The benchmark encodes videos using H.264 compression standard. This compression standard is widely used for recording, compression, and distribution of high definition video. H264_encode is a good representative of soft real-time applications. In addition to this, video encoding using H.264 standard is used for video streaming in high-performance mission-critical networks [20, 52, 80].

6.3.3 Experimental methodology

The experiments were designed in such a way as to provide reliable results and minimize the impact of the operating system processes. Each experiment was repeated 50 times and each time we measured the execution time of the application under study. As our study addresses timing predictability and timing bounds of applications, we reported and analyzed the Longest Observed Execution Time (LOET) in 50 repetitions. We measured the execution time of an experiment by reading the time stamp counter (*rdtsc*). The time stamp counter is a 64-bit register that counts the number of ticks since reset on x86 processors. We accessed the counter register using the macro written in x86 assembly that is included in the experimental framework. This way, we avoided any system call for measuring time.

As the experiments were executed on a full-fledged Linux operating system (OS), we paid special attention to minimizing the impact of the OS to our measurements [71, 122, 133]. To avoid task migration among different cores (virtual CPUs, hardware contexts, strands) of a processor, we bound each benchmark to the corresponding core using the *sched_setaffinity* system call.

In order to quantify the impact of the OS processes on the platforms used in the study, we repeated each benchmark in isolation for 10,000 times and measured the impact of

OS processes to the execution time of the benchmark. Our results show that the impact of interference with the OS processes to variation of benchmark execution time is below 1%.

6.4 Evaluation

In this section, we show how the proposed methodology can be applied to determine the suitability of the three MT COTS architectures presented in the previous section for time-critical environments. This scenario is relevant when companies have to determine which MT COTS architectures are good candidates to be used in their future time-critical system. After using our methodology to select a subset of processors with appropriate distribution and characteristics of shared resources, the company can do a detailed analysis of each of the selected processors to provide stronger guarantees that it meets the requirements of the system.

If an architecture under study offers a mechanism that could provide isolation between simultaneously-running threads (e.g. cache partitioning), the presented methodology can be used to characterize different configurations of a single architecture. None of the architectures under study provides a support for partitioning of shared hardware resources (thread isolation), thus we were not able to explore this approach. We believe that it is an interesting avenue of future work.

6.4.1 Potential execution time variation

We start by analyzing the potential slowdown that applications may experience because of the collision in shared processor resources. To that end, we run all resource-stressing benchmarks in isolation and in different workloads. By observing the slowdown of the benchmarks, we can quantify the potential slowdown a real application would experience due to resource sharing. The results are presented in Table 6.2. Each entry of the table shows the slowdown that the benchmark under study (listed in the rows of the table) experiences when it is simultaneously executed with the stressing benchmarks (columns).

Atom processor. When running on the Atom processor, all benchmarks in our resource-stressing benchmark suite have a co-runner that makes them experience significant slowdown, see Table 6.2(a). In most of the experiments, the benchmark under study experiences the highest slowdown when it is simultaneously executed with one more instance

Table 6.2: Possible slowdown because of the collision in processor resources

		Pipeline front end	Pipeline back end (Execution units)						Cache memory			Off-chip resources
		nop	intAdd	intMul	intDiv	fpAdd	fpMul	fpDiv	L1 data	L1 icache	L2	mem_bw
Pipeline front end	nop	2.0	2.0	2.0	1.8	1.6	1.7	1.6	1.7	1.8	1.7	1.6
Pipeline back end (Execution units)	intAdd	2.0	2.0	1.2	1.2	1.2	1.0	1.0	1.0	1.1	1.0	1.0
	intMul	1.3	1.3	1.3	1.1	1.1	1.1	1.0	1.2	1.2	1.1	1.0
	intDiv	1.2	1.2	1.2	1.2	1.1	1.1	1.8	1.2	1.1	1.2	1.1
	fpAdd	1.1	1.2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	fpMul	1.0	1.0	1.0	1.1	1.0	1.0	1.0	1.1	1.0	1.1	1.0
	fpDiv	1.0	1.0	1.0	1.5	1.0	1.0	2.0	1.0	1.0	1.0	1.0
Cache memory	L1 dcache	1.0	1.0	1.2	1.0	1.0	1.1	1.0	**6.2**	1.0	3.6	3.6
	L1 icache	1.1	1.1	1.0	1.0	1.0	1.0	1.0	1.0	**2.7**	1.0	1.0
	L2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	**2.3**	1.0	**14.1**	**15.3**
Off-chip resources	mem_bw	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	**2.5**	**2.5**

(a) Atom

		Pipeline front end	Pipeline back end (Execution units)						Cache memory			Off-chip resources
		nop	intAdd	intMul	intDiv	fpAdd	fpMul	fpDiv	L1 data	L1 icache	L2	mem_bw
Pipeline front end	nop	1.0	1.0						1.0			1.0
Pipeline back end (Execution units)												
Cache memory	L1 dcache											
	L1 icache											
	L2											1.1
Off-chip resources	mem_bw											1.3

(b) Pentium D

		Pipeline front end	Pipeline back end (Execution units)						Cache memory			Off-chip resources
		nop	intAdd	intMul	intDiv	fpAdd	fpMul	fpDiv	L1 data	L1 icache	L2	mem_bw
Pipeline front end	nop	1.0	1.0						1.0			1.0
Pipeline back end (Execution units)												
Cache memory	L1 dcache											
	L1 icache											
	L2										14.4	14.4
Off-chip resources	mem_bw										1.1	1.1

(c) Core2Quad: RS-nn distribution

		Pipeline front end	Pipeline back end (Execution units)						Cache memory			Off-chip resources
		nop	intAdd	intMul	intDiv	fpAdd	fpMul	fpDiv	L1 data	L1 icache	L2	mem_bw
Pipeline front end	nop	1.0	1.0						1.0			1.0
Pipeline back end (Execution units)												
Cache memory	L1 dcache											
	L1 icache											
	L2										1.3	1.3
Off-chip resources	mem_bw											

(d) Core2Quad: Rn-SS distribution

of the same resource-stressing benchmark. We also observe that, in general, the detected slowdown is quite high (up to 15.3×) and that the variation of the slowdown for different benchmarks is also very high. The slowdown due to the sharing of pipeline resources is less than 2×. When co-running tasks collide in the cache memory, the slowdown is higher and it ranges up to 6.2×, 2.7×, 14.1×, and 15.3× due to collision in the L1 data cache, instruction cache, L2 cache, and memory bandwidth, respectively. The mem_bw stressing benchmark stresses memory bandwidth through shared L2 cache. Hence, the slowdown that the benchmark under study experiences when it is co-scheduled with mem_bw stressing benchmark is the consequence of cumulative collision in the L2 cache and the memory bandwidth.

As applications running on the Atom processor may experience a significant slowdown because of interference in several resources, it is very difficult to estimate WCET of concurrently running time-critical tasks. Therefore, we conclude that the hyperthreading feature in the Atom processor should be avoided in time-critical environments that require predictability of application execution time. Disabling hyperthreading would lead to a more predictable execution time of the applications. However, this would make the Atom a single-threaded processor that would defy all performance improvements of MT architectures.

It is important to emphasize that our characterization does not conclude that the Atom is a poorly designed processor. Our experiments show that it is very difficult to estimate WCET of time-critical tasks concurrently running this processor, and, therefore, we do not recommend its use in time-critical environments. However, the Atom processor could be very suitable for systems that do not necessitate WCET analysis, i.e. the systems in which average performance, performance per watt, or performance per joule are the most important requirements.

Pentium D processor. The results for the Pentium D processor are presented in Table 6.2(b). We do not detect any slowdown because of the collision in execution pipeline resources (front-end of the pipeline, integer and FP execution units) or L1 data and instruction caches. These resources are private to each core, and, therefore, not shared among co-running tasks. Simultaneously-running applications only share the memory bandwidth. The slowdown we detect when the two mem_bw benchmarks simultaneously execute is very low, only 30%. We also detect interference between L2 and mem_bw benchmarks (10% slowdown). This is due to the fact that the L2 benchmark traverses an array whose size is equal to the size of the L2 cache. Although we try to access the whole

cache equally, the replacement policy may put a higher stress on some sets. This causes the L2 benchmark to experience some L2 misses. The memory accesses that miss in the L2 cache collide with mem_bw in the memory bandwidth.

To sum up, the potential slowdown because of inter-task interference on Pentium D processor is fairly low (around 30%). In addition to this, the co-running tasks do not interfere in most of the processor resources. Thus, it is much easier to estimate WCET of co-running time-critical tasks. We conclude that the architectures like Pentium D are good candidates to be used in time-critical environments that require predictability of application execution time.

Core2Quad processor. The Core2Quad processor has two levels of resource sharing [154], see Figure 6.3(c). In order to cover all possible scenarios, we analyzed the two benchmark distributions for the Core2Quad processor: (1) RS-nn, when the benchmark under study and the stressing benchmark share L2 cache, and (2) Rn-SS when the benchmark under study and stressing benchmarks do not share L2 cache, but only the memory bandwidth.

Results for the RS-nn distribution are presented in Table 6.2(c). We detect a significant slowdown when the two benchmarks collide in the L2 cache. The L2 resource-stressing benchmark experiences a slowdown of 14.4× when co-running with one more instance of L2 or mem_bw benchmark. In addition to this, we detect a 10% slowdown because of collision in memory bandwidth.

Results for the distribution in which the benchmark under study does not share the L2 cache with stressing co-runners are presented in Table 6.2(d). As for the Pentium D processor, we detect only a low interference in the memory bandwidth (30% slowdown). In addition to interference between mem_bw benchmarks, we also detect a slowdown when reference mem_bw is co-scheduled with two instances of the L2 benchmark. When two stressing L2 benchmarks share the L2 cache, they experience a lot of L2 cache misses that access the main memory and stress memory bandwidth, as the majority of the instructions miss in L2 cache. Therefore, although the L2 benchmark is not designed to stress bandwidth to the main memory, the two benchmark instances that share L2 cache are very bad co-runners for applications using memory bandwidth.

From the analysis of the Core2Quad processor, we see that the potential slowdown that the application experiences because of collision in processor resources with the co-running tasks also depends on the distribution of running benchmarks. When the two running benchmarks share L2 cache, they can experience a significant slowdown because

of the collision in this resource. However, if the benchmarks are distributed in such a way that they share only the memory bandwidth, the slowdown we detect is low, around 30%. Therefore, architectures similar to Core2Quad processor are good candidates for systems with timing requirements, as long as time-critical applications do not share the L2 cache memory with co-running tasks.

6.4.2 Application sensitivity to resource sharing

The results presented in the previous section show that applications simultaneously-running on a processor may experience high variations in execution time because of collision in shared processor resources.

Unlike stressing benchmarks, real applications do not use a single resource during their entire execution, but have different phases in which they use different resources. The usage patterns determine the actual effect that resource sharing has on the applications' execution time. In order to understand which resources are stressed by real applications, we execute them simultaneously with the resource-stressing benchmarks. We define the sensitivity of an application to a specific processor resource as the *slowdown that the application experiences when it is co-scheduled with corresponding resource-stressing benchmark*.

The results presented in Table 6.3 show the sensitivity of applications to resource sharing. Each entry of the table shows the slowdown that the real benchmark, whose sensitivity we measure (listed in the rows of the table), experiences when it is simultaneously executed with different resource-stressing benchmarks (columns).

Atom. The results for the Atom processor are presented in Table 6.3(a). We reach two conclusions: (1) All real benchmarks presented are sensitive to sharing most of the processor resources, and (2) The slowdown experienced by real applications when co-running with resource-stressing benchmarks is up to 22% for STAP Radar, 47% for H264_encode, and up to 18% for the CoreMark benchmark. This is significantly lower than the slowdown we detect for resource-stressing benchmarks (see Table 6.2(a)). This means that, although the underlying architecture has a strong potential to lead to high variation in the execution time of running applications, the particular real benchmarks that we used do not experience a significant slowdown.

Pentium D. The results for the Pentium D processor are presented in Table 6.3(b). Real benchmarks running on the Pentium D are only sensitive to sharing the memory

Table 6.3: Sensitivity of the real benchmarks to a collision in processor resources

	Pipeline front end	Pipeline back end (Execution units)						Cache memory			Off-chip resources
	nop	intAdd	intMul	intDiv	fpAdd	fpMul	fpDiv	L1 data	L1 icache	L2	mem_bw
STAP Radar	1.06	1.07	1.17	1.10	1.02	1.11	1.03	1.03	1.17	1.22	1.06
H264 encode	1.26	1.27	1.35	1.25	1.26	1.26	1.20	1.23	1.39	1.47	1.45
CoreMark	1.02	1.01	1.01	1.18	1.13	1.14	1.08	1.18	1.18	1.18	1.18

(a) Atom

	Pipeline front end	Pipeline back end (Execution units)						Cache memory			Off-chip resources
	nop	intAdd	intMul	intDiv	fpAdd	fpMul	fpDiv	L1 data	L1 icache	L2	mem_bw
STAP Radar	1.00	1.00						1.00			1.02
H264 encode											1.04
CoreMark											1.03

(b) Pentium D

	Pipeline front end	Pipeline back end (Execution units)						Cache memory			Off-chip resources
	nop	intAdd	intMul	intDiv	fpAdd	fpMul	fpDiv	L1 data	L1 icache	L2	mem_bw
STAP Radar	1.00	1.00						1.00		1.00	1.00
H264 encode										1.02	1.02
CoreMark										1.01	1.01

(c) Core2Quad: RS-nn distribution

	Pipeline front end	Pipeline back end (Execution units)						Cache memory			Off-chip resources
	nop	intAdd	intMul	intDiv	fpAdd	fpMul	fpDiv	L1 data	L1 icache	L2	mem_bw
STAP Radar	1.00	1.00						1.00		1.00	1.01
H264 encode											1.01
CoreMark											1.00

(d) Core2Quad: Rn-SS distribution

bandwidth. The slowdown that the real benchmarks experience is still lower than the slowdown of resource-stressing benchmarks (see Table 6.2(b)), but the difference is low.

Core2Quad. The results for the Core2Quad processor and the task distribution in which reference and stressing benchmarks share the L2 cache are presented in Table 6.3(c). We detect no slowdown for the STAP Radar benchmark, and a very low slowdown (below 2%) when H264_encode and CoreMark are executed with L2 and mem_bw co-runners. This means that the processor has the potential for significant slowdown because of a collision in the shared L2 cache, but that the given set of benchmarks is insensitive to this resource. However, if the target applications change, the possible variation in the execution time due to interference in L2 cache is high.

We repeat the experiments on the Core2Quad processor for the distribution in which real applications and resource-stressing benchmarks do not share the L2 cache, see Ta-

ble 6.3(d). We detect a very low slowdown (around 1%) when STAP Radar and H264_encode simultaneously execute with mem_bw stressing benchmark. Again, the slowdown that the real benchmarks experience is lower than the slowdown of resource-stressing benchmarks (see Table 6.2(d)).

One conclusion we reach from the presented results is that the real benchmarks we use in the study show low sensitivity to resource sharing. The other way to understand the results is that the applications under study are under-utilizing some of the processor resources (e.g. the L2 cache), so a less aggressive processor could be used to provide similar performance. It is important to note that these conclusions apply to the benchmarks and processors we analyze in the study and not to the proposed methodology. The same methodology applied to different real benchmarks or processors could reach different conclusions.

6.4.3 WCET estimation

In order to reduce costs, current and future real-time systems follow an integrated approach in which more functionality is executed on the same hardware. This requires hardware that provides higher performance, and that enables incremental timing verification. Incremental timing verification means that a user does not have to verify the timing behavior of all running applications each time a new component (application) is changed or added to the system [60]. In that sense, the system is time composable if the WCET estimate of the tasks do not change if any of the tasks in the workload change. In this section, we show how our methodology helps with providing composable WCET estimations.

When directly extending the the standard measurement-based approach (used in single-threaded processors) to MT architectures, the application under study should be executed with different sets of co-running tasks (in different workloads). The longest observed execution time of the application in any workload would then be used to estimate the WCET. We show that this approach does not properly quantify the impact of inter-task interaction to application execution time, and that can lead to an underestimation of WCET.

In the standard measurement-based analysis (Classical approach) the application under study is executed in different workloads. In our case, as we analyze three real benchmarks, we can run all possible workloads. In Table 6.4, we present the slowdown that the benchmarks under study (listed in the rows of the table) experience when they exe-

Table 6.4: Comparison of classical measurement-based timing analysis and our approach

	Classical approach				Our approach
	STAP Radar	H264 encode	CoreMark	Max	
STAP Radar	1.21	1.20	1.19	**1.21**	**1.22**
H264 encode	1.39	1.41	1.42	**1.42**	**1.47**
CoreMark	1.03	1.03	1.04	**1.04**	**1.18**

(a) Atom

	Classical approach				Our approach
	STAP Radar	H264 encode	CoreMark	Max	
STAP Radar	1.00	1.00	1.00	**1.00**	**1.02**
H264 encode	1.00	1.00	1.00	**1.00**	**1.04**
CoreMark	1.00	1.00	1.00	**1.00**	**1.03**

(b) Pentium D

	Classical approach				Our approach
	STAP Radar	H264 encode	CoreMark	Max	
STAP Radar	1.00	1.00	1.00	**1.00**	**1.00**
H264 encode	1.00	1.00	1.00	**1.00**	**1.02**
CoreMark	1.00	1.00	1.00	**1.00**	**1.01**

(c) Core2Quad: RS-nn distribution

	Classical approach				Our approach
	STAP Radar	H264 encode	CoreMark	Max	
STAP Radar	1.00	1.00	1.00	**1.00**	**1.01**
H264 encode	1.00	1.00	1.00	**1.00**	**1.01**
CoreMark	1.00	1.00	1.00	**1.00**	**1.00**

(d) Core2Quad: Rn-SS distribution

cute with different stressing benchmarks (columns). In the last column of the table (Our approach), we also present the slowdown of the benchmark that is computed by measuring benchmark sensitivity to inter-task interference in shared processor resources. This is the maximum slowdown that the benchmark experiences when it is co-scheduled with different resource stressing benchmarks (see Section 6.2.4.2).

Atom. The results for the Atom processor are presented in Table 6.4(a). For all three benchmarks, STAP Radar, H264 encode, and CoreMark, the maximum slowdown detected using the classical approach is exceeded in experiments with resource-stressing benchmarks. The difference between the slowdown measured by the classical approach and the slowdown measured using our methodology ranges up to 14% (CoreMark benchmark).

Pentium D. The results for the Pentium D processor are presented in Table 6.4(b). In this case, we detected no slowdown when pairs of real benchmarks executed on the

processor. However, when we execute STAP Radar, H264 encode, and CoreMark with resource stressing benchmarks, we detected a slowdown of up to 2%, 4%, and 3%, respectively. Again, the maximum slowdown detected when using the classical approach was exceeded in experiments with resource-stressing benchmarks.

Core2Quad. The results for the Core2Quad processor are presented in Tables 6.4(c) and (d). In both task distribution, RS-nn and Rn-SS, we detect no slowdown when workloads composed of real benchmarks execute on the processor. When H264 encode and CoreMark share the L2 cache with resource stressing benchmarks (see Table 6.4(c)) we detect slowdown of 2% and 1%, respectively. When STAP Radar and H264 encode share memory bandwidth with stressing benchmarks, the detected slowdown is 1%, see Table 6.4(d).

Real applications have different phases in which they stress different processor resources. Also, the stress that real applications put on each processor resource is not the highest possible stress of that resource. Therefore, experiments in which several real applications simultaneously execute on the processor are unlikely to capture the worst possible inter-task interference. Resource stressing benchmarks put high stress on specific processor resources. Running real applications with resource stressing benchmarks will detect inter-task interference in shared processor resources and properly quantify the impact of this interference on execution time. To summarize, measuring the interference between real applications and resource stressing benchmarks can significantly improve measurement-based methods for estimation of WCET in MT COTS processors.

As we explained in Section 6.2.1, the design of a worst-stressing benchmark is infeasible in practice. An interesting avenue of future work could be to analyze how the sensitivity of real applications to collision in different processor resources independently can be combined to estimate the worst possible slowdown that the application may experience because of interference with tasks co-running on the MT COTS processor. This value could be used to compute a good estimate of the application WCET independently from the set of co-running tasks.

6.4.4 Mixed stressing workloads

When the analyzed MT COTS processor comprises more than two cores or hardware contexts, the application under study can be co-scheduled with different sets of resource-stressing benchmarks. In these experiments, the slowdown that an application experiences

Table 6.5: The timing analysis of the Core2Quad processor: The improvement of the mixed stressing workload

	Homogeneous stressing workload			Mixed stressing workload			
	L2 L2 L2	mem_bw mem_bw mem_bw	Max	L2 L2 mem_bw	L2 mem_bw mem_bw	mem_bw L2 L2	mem_bw L2 mem_bw
STAP Radar	1.05	1.06	1.06	1.05	1.05	1.05	1.05
H264 encode	1.07	1.07	1.07	1.08	1.07	1.06	1.07
CoreMark	1.02	1.03	1.03	1.04	1.04	1.02	1.04

is a combination of collision in different processor resources, so this approach improves the estimation of application worst case slowdown.

In our study, Core2Quad is the only processor that supports simultaneous execution of more than two tasks. In order to test mixed criticality stressing workloads, we execute real benchmarks with homogeneous stressing workloads (three instances of L2 or mem_bw benchmark) and with workloads that combine L2 and mem_bw benchmarks. In these experiments, we use only L2 and mem_bw because they are the only benchmarks that stress shared resources of the Core2Quad processor (see Table 6.2). The results of the experiments with mixed stressing workloads are presented in Table 6.5. Each entry of the table shows the slowdown that the benchmark under study (listed in the rows of the table) experiences when it is simultaneously executed with different stressing workloads (columns). For two out of three real benchmarks under study, H264 encode and CoreMark, the slowdown caused by mixed stressing workload exceeds the highest slowdown of homogeneous workloads.

Overall, we show that running workloads composed of real applications may not be sufficient to determine which processor is a better candidate to be used in time-critical environments. In addition to analyzing the measured slowdown experienced by a given set of applications, it is also important to understand the potential slowdown that simultaneously-running applications can experience because of collision in shared resources. We show that the slowdown that applications experience because of collision in shared resources may be low if the applications are insensitive to these resources, even if the potential slowdown is very high.

6.4.5 Additional considerations

System level timing analysis: Once the presented methodology is used to quantify the slowdown that an application may experience due to inter-task interferences and WCET of the application is estimated, it is necessary to consider system level issues such as sharing of OS services, process preemption, context switching cost, or task scheduling, and to do a response time analysis. The impact of system level issues on an application WCET, and the response time analysis are out of the scope of the presented study.

Hybrid WCET analysis: Several studies propose *hybrid WCET analysis* as a method for the timing analysis of real-time systems. Hybrid WCET analysis is the combination of static program analysis and the measurement-based techniques [53, 93, 136, 159]. First, the hybrid WCET analysis statically analyzes the application code. As the analysis of all possible execution paths of real industrial programs is complex or even infeasible, hybrid WCET analysis divides the program into mutually exclusive segments and analyzes each segment separately. For each program segment, the analysis determines the different possible execution paths of the program and generates sets of input data that force the execution of each path. Later, the program is executed on real hardware for different input data sets provided by static analysis. For each set of input data, the user measures the execution time of the corresponding segment of the code. Finally, the WCET of the whole program is computed based on the measured execution time of different program segments. In hybrid WCET analysis, the measurements are performed on real hardware, so a detailed model of the architecture under study is not required. This is its main advantage when compared to the static WCET analysis. To the best of our knowledge, current hybrid WCET analysis studies are focused on WCET estimation of applications running on single-threaded processors. As a part of our future work, we plan to use the presented methodology to detect the possible slowdown of different program segments and to extend the current hybrid WCET analysis to MT COTS architectures.

In all the experiments, the slowdown that the benchmarks experience is measured from the entry until the termination of the benchmark execution (*end-to-end* measurements). However, the presented methodology can be easily adjusted to focus on different program segments, so it can be used in hybrid WCET analysis or when only some program segments have time-critical requirements. In this case, the set of resource-stressing benchmarks and the set of experiments would remain the same: the application under study should be executed in isolation and with the resource-stressing co-runners. The

only difference is that, in this case, instead of measuring the execution time (slowdown) of the whole application, the user should instrument only the program segments under study. Methods for low-overhead instrumentation of different application segments have already been proposed [135].

6.5 Related Work

Despite the benefits that MT COTS processors may offer in embedded real-time systems these architectures are still not widely used in real-time systems because the timing analysis is too complex. To the best of our knowledge, our study presents the first systematic approach for measurement-based timing analysis of time-critical applications running on MT COTS architectures. Several studies and projects analyze collision in shared hardware resources among tasks co-scheduled on MT architectures and the impact of this collision on WCET analysis.

Schliecker et al. [137] and Pellizzoni et al. [120] analyze the delay of memory access in systems where several simultaneously-running tasks share the main memory. The proposals require a detailed profiling of application memory access pattern and a deep understanding of the memory arbitration policy. Although both proposals can be applied only in systems where the main memory is the only shared resource (authors assume non-shared caches), we believe that these studies are a very good starting point to better understand the possible use of multithreaded processors in time-critical systems.

Cullmann et al. [49] analyze the design of future multithreaded processors for time-critical systems. The authors show that some processor designs make the timing analysis infeasible and suggest design principles for making multithreaded architectures predictable. This analysis is complementary to our study. Based on the theoretical analysis, the authors give guidelines for the design of predictable architectures, while we present the measurement-based approach to determine if a given architecture is a good candidate for time-critical systems.

Several projects propose hardware solutions to deal with the inter-task interferences on WCET in MT architectures [18, 68, 78, 110, 124, 125, 150]. These projects make a wide range of proposals. Some suggest preventing inter-task interferences by assigning each task a subset of resources and not allowing other user tasks to use the resources. This can be implemented by splitting the hardware resource temporally or spatially. Other proposals suggest actually allowing tasks to share hardware resources and defining the

boundaries of this interaction so that the maximum effect of the interaction on the WCET is known. Although these proposals are different, the common factor is that they all propose changes in hardware to reach their objectives. The objective of our study is to show how, without any change, MT COTS processors can assess the challenges and the requirements imposed in a real-time environment, mainly time predictability.

Several studies focus on Measurement-Based Timing Analysis (MBTA) for single-threaded architectures. The studies propose an improvement of the accuracy and coverage of MBTA by using static code analysis. Schaefer et al. [136] propose measuring execution time at basic block level and using this data to estimate WCET of the whole program. Deverge and Puaut [53] propose using structural testing methods to generate input data for experiments used in measurement-based WCET analysis. Kirner et al. [93] use static program analysis to generate test data that cover different execution paths. Authors also propose a decomposition of program paths into smaller parts (subpaths) and using an independent measurement-based analysis for each subpath. Finally, the WCET estimate of the whole program is calculated based on the execution time of each subpath. Wenzel et al. [159] present a similar approach: they propose a decomposition of the program into segments and doing a timing analysis for each segment. The authors also propose an approach for good program segmentation – one that balances the number of program segments with the average number of paths per segment. Rieder et al. [135] analyze different approaches for measuring the execution time of program segments. The authors propose an external Runtime Measurement Device and suggest the integration of this device into the analysis framework that automatically collects the data needed for measurement-based WCET analysis. All the above studies propose improvements of end-to-end measurement-based timing techniques for single-threaded processors. In our study, we extend measurement based timing analysis for multithreaded processors and show how it can be used to determine which architecture is more suitable for systems with timing requirements.

6.6 Summary

COTS processors are increasingly being considered in the design of systems with timing requirements. MT COTS architectures are of special interest due to good performance-per-watt ratio, high performance opportunities, and their suitability for embedded architectures in which several functions are integrated into the same processor.

Unfortunately, despite the benefits that MT COTS may offer in embedded real-time systems, the time-critical market has not yet embraced a shift toward these architectures. The main challenge with MT COTS architectures is the difficulty when predicting the execution time for simultaneosly-running time-critical tasks. Providing a timing analysis for real industrial applications running on MT COTS processors becomes extremely difficult because the execution time of a task, and hence its WCET depends on the interference with co-running tasks in shared processor resources.

In this chapter of the document, we have shown that the measurement-based timing analysis used for single-threaded processors cannot be directly extended for MT COTS architectures. Running workloads composed of real applications may not be sufficient to capture the possible slowdown that applications experience due to interferences. This is due to the fact that the applications used may be insensitive to interference in processor shared resources. We propose a methodology that quantifies the slowdown that a task may experience because of collision with co-runners in shared resources of MT COTS processor. To that end, we developed a set of specific resource-stressing benchmarks and propose a measurement-based approach to determine the possible slowdown caused by inter-task interferences. The two main applications of our methodology are: (1) The methodology helps to determine which architecture among different MT COTS processors is more suitable to be used in time-critical environments, (2) Our methodology shows the potential variation in execution time a task in a workload may experience if any of the co-runner changes.

We also presented several case studies in which we analyze three MT COTS architectures with different degrees of shared resources. We show that, for a given workload composed of several real-time benchmarks, all three types of architecture show low interference among co-running tasks and stable execution time. However, our method shows that potential variation in the execution time of applications is different for each architecture under study, and that not every one of the three architectures are good candidates to be used in time-critical systems.

Chapter 7

Conclusions

Multithreaded processors are a very good solution for exploiting processor performance beyond the limitations imposed by limited instruction-level parallelism and power wall. Currently, multithreaded architectures are the mainstream in the processor design. They are widely used in servers, desktop computers, lap-tops, and mobile devices.

However, multithreaded architectures introduce several challenges at the software level. In order to utilize the hardware potential optimally and to deliver maximum performance, state-of-the-art multithreaded architectures have to execute numerous software threads simultaneously. At the operating system level, one of the main challenges becomes how to schedule simultaneously-running threads. Also, the need for numerous threads motivates the development of multithreaded applications and algorithms. Development of efficient, portable, and correct multithreaded software requires novel programming models and paradigms, and increases the complexity of compilers. In time-critical environments, the timing analysis has to estimate the impact of the collision in shared hardware resources between simultaneously-running threads on the execution time of each thread.

This thesis presented cross-domain approaches that improve the effective use of multithreaded architectures. The contributions of the thesis can be classified in three groups. First, we proposed several methods for thread assignment of multithreaded network applications running in multithreaded servers. Second, we analyzed the problem of graph partitioning that is a part of the compilation process of multithreaded streaming applications. Finally, we presented a method that improves the measurement-based timing analysis of multithreaded architectures used in time-critical environments. The following sections briefly summarize each of the contributions.

7.1 Thesis contributions

7.1.1 Thread assignment on multithreaded processors

State-of-the-art multithreaded processors have different level of resource sharing (e.g. between thread running on the same core and globally shared resources). Therefore, the way that the threads of a given workload are assigned to processors' hardware contexts determines which resources the threads share, which, in turn, may significantly affect the system performance.

In this thesis, we demonstrated the importance of thread assignment for network applications running in multithreaded servers. We also presented *TSBSched* and *BlackBox* scheduler, simple methods for thread assignment of multithreaded network applications running on processors with several levels of resource sharing. We evaluated *TSBSched* and *BlackBox* scheduler for a set of network applications running on the UltraSPARC T2 processor. The presented results show very high accuracy of both proposed thread assignment methods, and significant performance improvement over random and Linux-like thread assignment approaches.

We also proposed a statistical approach to the thread assignment problem. In particular, we showed that running a sample of several hundred or several thousand random thread 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 thread 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 statistical 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 thread assignment of multithreaded network applications running on the UltraSPARC T2 processor. Our results show that running several thousand random thread assignments provided enough information for the precise estimation of the performance of the optimal thread assignment. Also, several thousand random thread assignments were 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.

7.1.2 Kernel partitioning of streaming applications

Another difficulty in modern computing systems is how to write efficient, portable, correct software for multithreaded processors. A promising approach is to expose more parallelism to the compiler through domain-specific languages, enabling the compiler to perform complex high-level transformations. One important application domain comprises stream programs. An important step in compiling a stream program to multiple processors is kernel partitioning, which significantly affects application performance. Finding an optimal kernel partition is, however, an intractable problem.

We proposed a statistical approach to the kernel partitioning problem. We described a method that statistically estimates, with a given confidence level, the performance of the optimal kernel partition. We demonstrated that the sampling method is an important part of the analysis, and that not all methods that generate *i.i.d.* samples provide good results. We also showed that random sampling on its own can be used to find a good kernel partition, and that it could be an alternative to heuristics-based approaches.

The presented statistical method does not depend on the application. It does not require any application profiling nor does it require the understanding of the application stream graph. The method can be applied to streaming applications with any number of kernels, and it can target any number of software threads. The presented method can analyze different metrics such as throughput, maximum hardware utilization, and minimum energy or power consumption.

We successfully applied the presented statistical analysis to the benchmarks included in the StreamIt 2.1.1 suite. The method precisely estimated the optimal kernel partition performance for all the benchmarks under study. Also, in our experiments, several hundred or several thousand random kernel partitions were enough to find a partition with close to optimal performance. The performance of the kernel partitions that were selected using random sampling matched the performance provided by the complex heuristic-based approach.

7.1.3 Multithreaded processors in time-critical environments

Commercial-off-the-shelf (COTS) processors are increasingly being considered in the design of systems with timing requirements. Multithreaded (MT) COTS architectures are of special interest due to good performance-per-watt ratio, high performance opportunities, and their suitability for embedded architectures in which several functions are integrated into the same processor.

Unfortunately, despite the benefits that MT COTS may offer in embedded real-time systems, the time-critical market has not yet embraced a shift toward these architectures. The main challenge with MT COTS architectures is the difficulty when predicting the execution time for simultaneously-running time-critical tasks. Providing a timing analysis for real industrial applications running on MT COTS processors becomes extremely difficult because the execution time of a task, and hence its worst-case execution time (WCET) depends on the interference with co-running tasks in shared processor resources.

We showed that the measurement-based timing analysis used for single-threaded processors cannot be directly extended for MT COTS architectures. Running workloads composed of real applications may not be sufficient to capture the possible slowdown that applications experience due to interferences. This is due to the fact that the applications used may be insensitive to interference in processor shared resources. We proposed a methodology that quantifies the slowdown that a task may experience because of collision with co-runners in shared resources of MT COTS processor. To that end, we developed a set of specific resource-stressing benchmarks and propose a measurement-based approach to determine the possible slowdown caused by inter-task interferences.

We also presented several case studies in which we analyze three MT COTS architectures with different degrees of shared resources. We show that, for a given workload composed of several real-time benchmarks, all three types of architecture show low interference among co-running tasks and stable execution time. However, our method shows that potential variation in the execution time of applications is different for each architecture under study, and that not all the architectures under study are good candidates to be used in time-critical systems.

7.2 Future work

7.2.1 Thread assignment

In this thesis, we addressed the problem of thread assignment on multithreaded network servers. Thread assignment is an important scheduling problem for state-of-the-art processors in general, and one of the most promising ways to increase the performance of data centers, high performance computing (HPC) clusters, and communication base stations.

We focused on the thread assignment problem of applications running on a single processor. However, the proposed thread assignment approaches can be applied to thread assignment (thread distribution) problem on different levels: between different processors of the same server, different servers in the rack, or between different racks in a data center.

The presented thread assignment approaches were designed for scheduling of network applications that are executed on multithreaded servers. It would be interesting to evaluate whether the presented thread assignment approaches could be used in different domains in which workload changes infrequently. Some of the target environments could be communication base stations, telecommunication servers, or front-end of data centers.

Although the presented thread assignment approaches are not designed to address the problem of dynamic process scheduling, random sampling and the statistical thread assignment approach can be used to evaluate heuristic-based approaches that target highly dynamic environments. The estimation of the possible performance improvement of dynamic heuristic-based thread assignment approaches can be done before the system deployment. Since this analysis can be done off-line, it is feasible to execute thousands of random thread assignments of a given workload, and to apply the presented statistical analysis to estimate the performance of the optimal thread assignment. Also, the heuristic-based thread assignment approaches can be evaluated for numerous application sets of interest.

In network servers, data centers and HPC clusters, the presented statistical analysis can be applied to different variations of job scheduling problems such as multiprocessor scheduling, resource-constrained scheduling, scheduling with individual deadlines, preemptive scheduling, priority scheduling, etc. The presented method is particularly well suited for systems that require analysis of different metrics such as throughput, fairness, hardware utilization, energy or power consumption, or any weighted sum of them.

Application of the presented thread assignment approaches to different scheduling problems in various domains that use diverse hardware platforms and different target metrics is an interesting avenue for future work.

7.2.2 Kernel (graph) partitioning

Several metrics can be used to describe the performance of streaming applications. In this thesis, we analyzed the cost of streaming applications – the time needed to process a fixed amount of input data. However, the proposed statistical approach is independent of the target metric, and it can be used to analyze different metrics of interest such as energy or power, the hardware utilization of the target architecture, or some weighted sum of them.

When the program is described using a stream language, the compiler may perform complex optimizations over the stream graph; it can combine adjacent kernels, split computationally intensive kernels into multiple parts, or duplicate kernels to have more parallel computation. In order to find the set of optimizations that provides the best performance, it is important to determine good kernel partitions for different optimization sets. In this case, the proposed kernel partitioning approach does not change, but the random sampling and the presented statistical analysis are simply repeated for different optimization sets, i.e. for different stream graphs of the same program.

In computer science, numerous problems are modeled in the form of graphs. For example, graph theory is highly utilized in mobile, ad-hoc, and computer networks, but also in data mining, clustering, image capturing and segmentation [51]. Many problems related to graph theory, such as graph covering, partitioning, coloring, and vertex ordering are intractable.

As a part of future work, we plan to apply the presented statistical analysis to different problems that are modeled by using graph theory. We plan to analyze these problems for different optimization sets and different metrics of interest.

7.2.3 Statistical analysis

Numerous problems in computer science are intractable. It is formally proven that many problems related to network design, program optimization, data storage and retrieval, process scheduling, graph and automata theory, are NP-complete [67]. Also, since many problems have a vast exploration space, it is unfeasible to do an exhaustive search in order to find a solution with the optimal performance.

Currently, intractable problems in computer science are usually addressed by using different heuristics-based approaches. The heuristics-based approaches are designed to address a specific problem and metric under study. Design of heuristics-based approaches and their adjustments to different problems or metrics requires significant effort.

The statistical approach that is presented in the thesis is easily applicable to different intractable problems and different metrics under study. The presented method does not require a profound understanding of the target system, which significantly reduces the effort and the time needed for a system deployment. In order to apply the statistical approach, the user should generate random samples and measure the performance of each of them. The statistical analysis that infers the population maximum (or minimum) based on the sample of measurements is independent of the problem that is addressed. Moreover, as demonstrated in the thesis, random sampling itself can be an effective approach to find good solutions for different problems. Application of random sampling and the presented statistical analysis to different problems in computer science is an interesting avenue of future work.

In order to estimate the optimal system performance, we used statistical analysis that is based on Extreme Value Theory (EVT). When dealing with the extreme deviations from the median of the probability distributions, EVT is a fundamental and indispensable statistical tool. Although EVT provides many theorems and tools that could be of interest to the community, currently, its application in computer science is marginal. As a part of future work, we plan to study EVT and to explore new applications of this branch of statistics in the field of computer science.

7.2.4 Timing analysis of multithreaded processors

The method for timing analysis that is presented in this thesis obtains a good estimation of the worst-case slowdown that real applications may experience because of collision in each of shared processor resources *independently*. As a part of future work, we plan to understand how the sensitivity of real applications to collision in different processor resources independently can be combined to estimate the overall worst possible slowdown that the application may experience because of interference with co-running tasks. This value could be used to compute a good estimate of the application WCET independently of the tasks that execute concurrently on multithreaded architectures.

In all the experiments presented in the thesis, the slowdown that the benchmarks experience is measured from the entry until the termination of the benchmark execution

(*end-to-end* measurements). Several studies (see Section 6.4.5) propose *hybrid WCET analysis* as a method for the timing analysis of real-time systems. Hybrid WCET analysis is the combination of static program analysis and the measurement-based techniques. To the best of our knowledge, current hybrid WCET analysis studies are focused on WCET estimation of applications running on single-threaded processors. As a part of our future work, we plan to use the timing analysis method that is presented in the thesis to extend the existing hybrid WCET approaches to multithreaded architectures.

7.3 Publications

In this section, we present a list of our research articles that are accepted for publication at conferences and journals. We also list the titles of the posters that are used to present our work at conferences and forums, and publications on other topics that are not considered to be the contributions of the thesis.

7.3.1 Conferences

- Petar Radojković, Paul M. Carpenter, Miquel Moretó, Alex Ramirez and Francisco Cazorla. *Kernel Partitioning of Streaming Applications: A Statistical Approach to an NP-complete Problem*. In Proceedings of 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45). Vancouver, Canada. December 2012.

- Petar Radojković, Vladimir Čakarević, Miquel Moretó, Javier Verdú, Alex Pajuelo, Francisco J. Cazorla, Mario Nemirovsky and Mateo Valero. *Optimal Task Assignment in Multithreaded Processors: A Statistical Approach*. In Proceedings of 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-2012). London, UK. March 2012.

  - The article is published also in ACM SIGARCH Computer Architecture News 40 (1), March 2012.

- Petar Radojković, Vladimir Čakarević, Javier Verdú, Alex Pajuelo, Francisco J. Cazorla, Mario Nemirovsky and Mateo Valero. *Thread to Strand Binding of Parallel Network Applications in Massive Multi-threaded Systems*. In Proceedings of 15th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP-2010). Bangalore, India. January 2010.

- The article is published also in ACM SIGPLAN Notices 45 (5), May 2010.

7.3.2 Journals

- Petar Radojković, Vladimir Čakarević, Javier Verdú, Alex Pajuelo, Francisco J. Cazorla, Mario Nemirovsky and Mateo Valero. *Thread Assignment of Multithreaded Network Applications in Multicore/Multithreaded Processors*. In IEEE Transactions on Parallel and Distributed Systems (TPDS). To appear in 2013.

- Petar Radojković, Sylvain Girbal, Arnaud Grasset, Eduardo Quiñones, Sami Yehia and Francisco J. Cazorla. *On the Evaluation of the Impact of Shared Resources in Multithreaded COTS Processors in Time-Critical Environments*. In ACM Transactions on Architecture and Code Optimization (TACO) 8 (4). January 2012.

7.3.3 Posters

- Petar Radojković, Francisco J. Cazorla, Javier Verdú, Alex Pajuelo, Mario Nemirovsky. *Thread Assignment of Network Applications in Multithreaded Processors: A Statistical Approach*. In EDAA/ACM SIGDA PhD Forum at Design, Automation, & Test in Europe (DATE 2013). Grenoble, France. March 2013.

- Petar Radojković, Paul M. Carpenter, Miquel Moretó, Alex Ramirez and Francisco Cazorla. *Kernel Partitioning of Streaming Applications: A Statistical Approach to an NP-complete Problem*. In 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45). Vancouver, Canada. December 2012.

- Petar Radojković, Francisco J. Cazorla, Javier Verdú, Alex Pajuelo, Mario Nemirovsky. *Thread Assignment of Network Applications in Multithreaded Processors: A Statistical Approach*. In Proceedings of 3rd Barcelona Forum on Ph.D. Research in Information and Communication Technologies. Barcelona, Spain. October 2012.

- Petar Radojković, Vladimir Čakarević, Javier Verdú, Alex Pajuelo, Roberto Gioiosa, Francisco J. Cazorla, Mario Nemirovsky and Mateo Valero. *Measuring Operating System Overhead on Sun UltraSPARC T1 Processor*. In Fifth International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES 2009). Terrasa (Barcelona), Spain. July 2009.

7.3.4 Other publications

- Vladimir Čakarević, Petar Radojković, Javier Verdú, Alex Pajuelo, Francisco J. Cazorla, Mario Nemirovsky and Mateo Valero. *Characterizing the Resource Sharing Levels in the UltraSPARC T2 Processor*. In Proceedings of 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42). New York City, US. December, 2009.

- Petar Radojković, Vladimir Čakarević, Javier Verdú, Alex Pajuelo, Roberto Gioiosa, Francisco J. Cazorla, Mario Nemirovsky and Malero Valero. *Measuring Operating System Overhead on CMT Processors*. In Proceedings of 20th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). Campo Grande, Brazil. October, 2008.

- Vladimir Čakarević, Petar Radojković, Javier Verdú, Alex Pajuelo, Roberto Gioiosa, Francisco J. Cazorla, Mario Nemirovsky and Malero Valero. *Understanding the Overhead of the Spin-lock Loop in CMT Architectures*. In Proceedings of 8th Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA), in conjunction with ISCA-35. Beijing, China. June 2008.

- Vladimir Čakarević, Petar Radojković, Javier Verdú, Alex Pajuelo, Roberto Gioiosa, Francisco J. Cazorla, Mario Nemirovsky and Malero Valero. *Overhead of the Spin-lock Loop in UltraSPARC T2*. In Proceedings of 5th HiPEAC Industrial Workshop: Tools and Methodology for Parallel Programming. Barcelona, Spain. June 2008.

7.4 Awards

The **HiPEAC Paper Award**[1] aims to encourage HiPEAC members to publish their work at conferences in which Europe is not strongly represented. The following publications received the HiPEAC Paper Award:

- Petar Radojković, Paul M. Carpenter, Miquel Moretó, Alex Ramirez and Francisco Cazorla. *Kernel Partitioning of Streaming Applications: A Statistical Approach to an NP-complete Problem*. In Proceedings of 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45). Vancouver, Canada. December 2012.

[1] More information about the HiPEAC Paper Award could be found at *http://www.hipeac.net/award*.

- Petar Radojković, Vladimir Čakarević, Miquel Moretó, Javier Verdú, Alex Pajuelo, Francisco J. Cazorla, Mario Nemirovsky and Mateo Valero. *Optimal Task Assignment in Multithreaded Processors: A Statistical Approach*. In Proceedings of 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-2012). London, UK. March 2012.

- Vladimir Čakarević, Petar Radojković, Javier Verdú, Alex Pajuelo, Francisco J. Cazorla, Mario Nemirovsky and Mateo Valero. *Characterizing the Resource Sharing Levels in the UltraSPARC T2 Processor*. In Proceedings of 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42). New York City, US. December, 2009.

Bibliography

[1] Intel® 64 and IA-32 Architectures Software Developer's Manual. Intel Corporation 2011.

[2] *StreamIt Language Specification, Version 2.1*. MIT Computer Architecture Group, 2006.

[3] StreamIt project. http://groups.csail.mit.edu/cag/streamit/.

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

[5] The CAIDA Anonymized Internet Traces Dataset. Cooperative Association for Internet Data Analysis - CAIDA. San Diego Supercomputer Center, University of California, San Diego. http://www.caida.org/.

[6] *OpenSPARC^{TM} T1 Microarchitecture Specification*. Sun Microsystems, Inc, 2006.

[7] *UltraSPARC T1^{TM} Supplement to the UltraSPARC Architecture 2005*. Sun Microsystems, Inc, 2006.

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

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

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

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

[12] IBM Power Systems Announcement Overview, 2010. http://www-03.ibm.com/systems/kw/resources/systems_power_news_20100209_annc.pdf.

[13] Oracle's SPARC T4-1, SPARC T4-2, SPARC T4-4, and SPARC T4-1B Server Architecture. *Oracle White Paper*, 2012.

[14] Intel 64 and IA-32 Architectures Software Developer's Manual, Intel Corporation, 2009. http://www.intel.com/Assets/PDF/manual/253665.pdf.

[15] J. Aas. Understanding the Linux 2.6.8.1 CPU Scheduler. *Silicon Graphics, Inc. (SGI)*, 2005.

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

[17] ACOTES. IST ACOTES Project Deliverable D2.2 Report on Streaming Programming Model and Abstract Streaming Machine Description Final Version, 2008.

[18] ACROSS. ARTEMIS CROSS-Domain Architecture. http://www.across-project.eu.

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

[20] F. Andritsopoulos, S. Papastefanos, G. Georgakarakos, and G. Doumenis. Reliable multicast H.264 video streaming for surveillance applications. *IEEE 18th International Symposium on Personal, Indoor and Mobile Radio Communications*, 2007.

[21] Atom Z530. Intel® Atom™ Processor Z5xx Series, 2009. http://download.intel.com/design/processor/datashts/319535.pdf.

[22] AUTOSAR. AUTomotive Open System ARchitecture. http://www.autosar.org.

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

[24] T. Baker. Lessons learned integrating COTS into systems. In *COTS-Based Software Systems*, Lecture Notes in Computer Science, 2002.

[25] A. A. Balkema and L. de Haan. Residual life time at great age. *Annals of Probability*, 2, 1974.

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

[27] C. Boneti, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, C.-Y. Cher, and M. Valero. Software-Controlled Priority Characterization of POWER5 Processor. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA)*, 2008.

[28] D. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly Media, Inc., 2006.

[29] J. V. Bradley. *Distribution-Free Statistical Tests*. Prentice-Hall, 1968.

[30] I. Buck. Brook Spec v0.2, 2003.

[31] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Professional, 1997.

[32] P. M. Carpenter, A. Ramirez, and E. Ayguade. Mapping stream programs onto heterogeneous multiprocessor systems. In *Proceedings of the international conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2009.

[33] P. M. Carpenter, D. Rodenas, X. Martorell, A. Ramirez, and E. Ayguadé. A streaming machine description and programming model. *Proceedings of the International Symposium on Systems, Architectures, Modeling and Simulation*, 2007.

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

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

[36] F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernández, A. Ramirez, and M. Valero. Architectural support for real-time task scheduling in SMT processors.

In *Proceedings of the 2005 international conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2005.

[37] F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero. Predictable Performance in SMT Processors: Synergy between the OS and SMTs. *IEEE Transactions on Computers*, 55(7), 2006.

[38] F. J. Cazorla, A. Ramirez, M. Valero, E. Fernández, and P. G. Canaria. Dynamically Controlled Resource Allocation in SMT Processors. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, 2004.

[39] F. J. Cazorla, A. Ramirez, M. Valero, P. M. Knijnenburg, R. Sakellariou, and E. Fernandez. QoS for High-Performance SMT Processors in Embedded Systems. *IEEE Micro*, 24(4), 2004.

[40] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA)*, 2005.

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

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

[43] F. L. Chevalier and S. Maria. STAP processing without noise-only reference: requirements and solutions. *International Conference on Radar*, 2006.

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

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

[46] Core2Quad. Intel® Core™2Quad Extreme Processor QX9000 and Intel® Core™Quad Processor Q9000 Series Datasheet. http://www.intel.com/design/processor/datashts/318726.htm?wapkw=(datasheet+q9000).

[47] CoreMark. The Embedded Microprocessor Benchmark Consortium benchmark suite. http://www.coremark.org.

[48] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quinones, and F. J. Cazorla. Measurement-based probabilistic timing analysis for multi-path programs. In *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems*, 2012.

[49] C. Cullmann, C. Ferdinand1, G. Gebhard, D. Grund, C. Maiza, J. Reineke, B. Triquet, and R. Wilhelm. Predictability considerations in the design of multi-core embedded systems. In *Embedded Real Time Software and Systems (ERTS)*, 2010.

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

[51] N. Deo. *Graph theory with applications to engineering and computer science*. Prentice Hall, 1974.

[52] A. Detti, P. Loreti, N. Blefari-Melazzi, and F. Fedi. Streaming H.264 scalable video over data distribution service in a wireless environment. *International Symposium on A World of Wireless, Mobile and Multimedia Networks*, 2010.

[53] J.-F. Deverge and I. Puaut. Safe measurement-based WCET estimation, 2005.

[54] D. Doucette and A. Fedorova. Base Vectors: A Potential Technique for Microarchitectural Classification of Applications. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*, 2007.

[55] M. Duranton, S. Yehia, B. D. Sutter, K. D. Bosschere, A. Cohen, B. Falsafi, G. Gaydadjiev, M. Katevenis, J. Maebe, H. Munk, N. Navarro, A. Ramirez, O. Temam, and M. Valero. *The HiPEAC Vision*. Network of Excellence on High Performance and Embedded Architecture and Compilation, 2011.

[56] D. Eckhoff, T. Limmer, and F. Dressler. Hash Tables for Efficient Flow Monitoring: Vulnerabilities and Countermeasures. In *Proceedings of the 34th Conference on Local Computer Networks (LCN)*, 2009.

[57] EEMBC. The Embedded Microprocessor Benchmark Consortium benchmark suite. http://www.eembc.org.

[58] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity–the Ptolemy approach. *Proceedings of the IEEE*, 91(1), 2003.

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

[60] EMPRESS. Incremental Verification and Validation Practices, EMPRESS Public Deliverable. http://www.empress-itea.org/.

[61] S. Eyerman and L. Eeckhout. Per-thread cycle accounting in SMT processors. In *Proceeding of the 14th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[62] S. Eyerman and L. Eeckhout. Probabilistic job symbiosis modeling for SMT processor scheduling. In *Proceeding of the 15th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[63] S. M. Farhad, Y. Ko, B. Burgstaller, and B. Scholz. Orchestration by approximation: mapping stream programs onto multicore architectures. In *Proceedings of the sixteenth international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[64] A. Fedorova, M. Seltzer, and M. Smith. Improving performance isolation on chip multiprocessors via an operating systems scheduler. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2007.

[65] W. Feller. *An introduction to Probability Theory and Its Applications*. John Wiley & Sons, Inc., 1971.

[66] J. E. Fritts, F. W. Steiling, and J. A. Tucek. Mediabench II video: Expediting the next generation of video systems research.

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

[68] GENESYS. GENeric Embedded SYStem Platform. http://www.genesys-platform.eu.

[69] G. Gereffi. *A Commodity Chains Framework for Analyzing Global Industries*. Duke University, USA, 1999.

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

[71] R. Gioiosa, F. Petrini, K. Davis, and F. Lebaillif-Delamare. Analysis of system overhead on parallel computers. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing (SC)*, 2003.

[72] M. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[73] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. Meli, A. Lamb, C. Leger, J. Wong, and H. Hoffmann. A Stream Compiler for Communication-Exposed Architectures. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[74] M. Greenwood. The natural duration of cancer. *Reports on Public Health and Medical Subjects*, 33, 1926.

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

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

[77] S. Ha and E. Lee. Compile-time scheduling and assignment of data-flow program graphs with data-dependent iteration. *IEEE Transactions on Computers*, 40(11), 1991.

[78] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken. Compsoc: A template for composable and predictable multi-processor system on chips. *Transactions on Design Automation of Electronic Systems (TODAES)*, 2009.

[79] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2012.

[80] K. J. Higgins. Video IP project boosts networks profile. *Network Computing*, 2004.

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

[82] G. Houston. BGP Table Statistics. http://bgp.potaroo.net.

[83] ILOG. CPLEX Math Programming Engine. http://www.ilog.com/products/cplex/.

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

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

[86] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer. Adaptive insertion policies for managing shared caches. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[87] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proceedings of the 17th international conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

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

[89] 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.

[90] S. M. Khan, D. A. Jiménez, D. Burger, and B. Falsafi. Using dead blocks as a virtual victim cache. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques (PACT)*, 2010.

[91] 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.

[92] R. Kirner and P. Puschner. Obstacles in Worst-Case execution time analysis. In *Proceedings of the 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, 2008.

[93] R. Kirner, P. Puschner, and I. Wenzel. Measurement-based worst-case execution time analysis using automatic test-data generation. In *Proceedings of IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, 2004.

[94] R. Kirner, I. Wenzel, B. Rieder, and P. Puschner. Using Measurements as a Complement to Static Worst-Case Execution Time Analysis. In *Intelligent Systems at the Service of Mankind*, volume 2. UBooks Verlag, Dec. 2005.

[95] E. Kohler, J. Li, V. Paxson, and S. Shenker. Observed Structure of Addresses in IP Traffic. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurements, Marseille, France*, 2002.

[96] 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.

[97] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design & Impl.*, 2008.

[98] 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 *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, 2004.

[99] Y.-K. Kwok and I. Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3), Dec. 1999.

[100] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4), Dec. 1999.

[101] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. IBM POWER6 microarchitecture. *IBM Journal of Research and Development*, 51(6), 2007.

[102] D. Levin, Y. Peres, and E. Wilmer. *Markov chains and mixing times*. American Mathematical Society, 2009.

[103] S. Liao, Z. Du, G. Wu, and G. Lueh. Data and Computation Transformations for Brook Streaming Applications on Multiprocessors. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2006.

[104] L. Lovász. Random walks on graphs: A survey. *Combinatorics, Paul Erdos is Eighty*, 2(1), 1993.

[105] W. Lundgren, K. Barnes, and J. Steed. Gedae: Auto Coding to a Virtual Machine. In *8th High Performance Embedded Computing Workshop*, 2004.

[106] C. Luque, M. Moreto, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, and M. Valero. CPU Accounting in CMP Processors. *in IEEE Computer Architecture Letters*, 8(1), 2009.

[107] C. Luque, M. Moreto, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, and M. Valero. ITCA: Inter-task Conflict-Aware CPU Accounting for CMPs. In *Proceedings of the 18th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2009.

[108] R. Manikantan, K. Rajan, and R. Govindarajan. NUcache: An efficient multicore cache organization based on Next-Use distance. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2011.

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

[110] MERASA. Multi-Core Execution of Hard Real-Time Applications Supporting Analysability. http://www.merasa.org.

[111] E. Mezzetti and T. Vardanega. On the Industrial Fitness of WCET Analysis. In *Proceedings of the 11th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2011.

[112] G. E. Moore. Cramming more components onto integrated circuits (Cramming more components onto integrated circuit for improved reliability and cost). *Electronics*, (38), 1965.

[113] M. D. Natale and A. Sangiovanni-Vincentelli. Moving from federated to integrated architectures in automotive: The role of standards, methods and tools. *Proceedings of the IEEE*, 2010.

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

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

[116] K. Olukotun and L. Hammond. The future of microprocessors. *Queue*, 3(7), Sept. 2005.

[117] M. Paolieri. *A Multi-core Processor for Hard Real-Time Systems, PhD Thesis*. Universitat Politècnica de Catalunya, 2011.

[118] M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero. Hardware Support for WCET Analysis of Hard Real-Time Multicore Systems. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, 2009.

[119] M. Paolieri, E. Quinones, F. J. Cazorla, and M. Valero. An analyzable memory controller for hard real-time CMPs. In *Embedded System Letters*, 2009.

[120] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2010.

[121] Pentium D. Intel® Pentium® D Processor 900 Sequence and Intel® Pentium® Processor Extreme Edition 955, 965, 2007. http://www.intel.com/Assets/PDF/datasheet/310306.pdf.

[122] 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 *Proceedings of the 2003 ACM/IEEE conference on Supercomputing (SC)*, 2003.

[123] J. I. Pickands. Statistical inference using extreme value order statistics. *Annals of Statistics*, 3, 1975.

[124] PREDATOR. PREDATOR consortium. http://www.predator-project.eu.

[125] PRET. Precision Timed (PRET) Machines. http://chess.eecs.berkeley.edu/pret.

[126] P. Puschner and A. Burns. A review of worst-case execution-time analysis. *Journal of Real-Time Systems*, 18(2/3), May 2000.

[127] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2006.

[128] 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 *Proceedings of the 45th International Symposium on Microarchitecture (MICRO-45)*, 2012.

[129] P. Radojković, S. Girbal, A. Grasset, E. Quiñones, S. Yehia, and F. J. Cazorla. On the Evaluation of the Impact of Shared Resources in Multithreaded COTS Processors in Time-critical Environments. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4), Jan. 2012.

[130] 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 *Proceedings of the 17th International Conference on*

Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2012.

[131] 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 *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2010.

[132] 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 Transactions on Parallel and Distributed Systems (TPDS)*, 2013.

[133] 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 *Proceedings of the 20th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2008.

[134] J. M. Richard McDougall. *Solaris internals: Solaris 10 and OpenSolaris kernel architecture*. Sun Microsystems Press/Prentice Hall, 2006.

[135] B. Rieder, I. Wenzel, K. Steinhammer, and P. Puschner. Using a runtime measurement device with measurement-based WCET analysis. *In Proceedings of International Embedded Systems Symposium (IESS)*, 2007.

[136] S. Schaefer, B. Scholz, S. M. Petters, and G. Heiser. Static analysis support for measurement-based WCET analysis. In *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2006.

[137] S. Schliecker, M. Negrean, G. Nicolescu, P. Paulin, and R. Ernst. Reliable performance analysis of a multicore multithreaded system-on-chip. In *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, CODES+ISSS, 2008.

[138] J. Sermulins, W. Thies, R. Rabbah, and S. Amarasinghe. Cache aware optimization of stream programs. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, Compilers, and Tools for Embedded Systems*, 2005.

[139] 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.

[140] T. Sherwood, G. Varghese, and B. Calder. A Pipelined Memory Architecture for High Throughput Network Processors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, 2003.

[141] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley, 2012.

[142] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. *IBM Journal of Research and Development*, 49(4/5), 2005.

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

[144] 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.

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

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

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

[148] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. *International Conference on Compiler Construction*, 2002.

[149] L. A. Torrey, J. Coleman, and B. P. Miller. A comparison of interactivity in the Linux 2.6 scheduler and an MLFQ scheduler. *Software - Practice and Experience*, 37(4), 2007.

[150] TTA. Time-Triggered Architecture. http://www.vmars.tuwien.ac.at/projects/tta.

[151] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *Proceedings of the 34th International Symposium on Microarchitecture (MICRO)*, 2001.

[152] T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quinones, M. Gerdes, M. Paolieri, and J. Wolf. Merasa: Multi-core execution of hard real-time applications supporting analysability. *IEEE Micro*, 2010.

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

[154] 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 *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

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

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

[157] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, et al. Baring It All to Software: Raw Machines. *Computer*, 1997.

[158] C. Watkins and R. Walter. Transitioning from federated avionics architectures to Integrated Modular Avionics. *In Proceedings of 26th Digital Avionics Systems Conference (DASC)*, 2007.

[159] I. Wenzel, R. Kirner, B. Rieder, and P. Puschner. Measurement-based timing analysis. In *Proceedings of 3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, 2008.

[160] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, May 2008.

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

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

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

[164] T. Wolf, N. Weng, and C.-H. Tai. Design considerations for network processor operating systems. In *Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communication Systems (ANCS)*, 2005.

[165] Y. Xie and G. H. Loh. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, 2009.

[166] D. Zhan, H. Jiang, and S. C. Seth. STEM: Spatiotemporal Management of Capacity for Intra-core Last Level Caches. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010.

[167] D. Zhan, H. Jiang, and S. C. Seth. Locality & utility co-optimization for practical capacity management of shared last level caches. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS)*, 2012.

List of Figures

List of Tables

Glossary

ACET Average Case Execution Time.

AUTOSAR Automotive Open System Architecture.

CDF Cumulative Distribution Function.

CMP Chip Multiprocessor.

COTS Commercial Off-The-Shelf.

CPU Central Processing Unit.

DFS Depth-First Search.

DRAM Dynamic Random Access Memory.

EC Edge Contraction.

EC-F Edge Contraction with Filter.

ECDF Empirical Cumulative Distribution Function.

EEMBC Embedded Microprocessor Benchmark Consortium.

EVT Extreme Value Theory.

FGMT Fine-Grain Multithreading.

FP Floating Point.

FPU Floating Point and Graphic Unit.

GPD Generalized Pareto Distribution.

GUI Graphical User Interface.

HWPipe Hardware Pipeline.

i.i.d. Independent and Identically Distributed.

I/O Input/Output.

IEU Integer Execution Unit.

IFU Instruction Fetch Unit.

ILP Instruction Level Prallelism.

IMA Integrated Modular Avionics.

IPFwd IP Forwarding.

IPv4 Internet Protocol version 4.

IT Information Technology.

LOET Longest Observed Execution Time.

LSU Load Store Unit.

MBTA Measurement-Based Timing Analysis.

MPPS Millions of Packets Per Second.

MT Multithreaded.

NA Not Applicable.

NIU Network Interface Unit.

NTGen Network Traffic Generator.

OS Operating System.

P Processing thread of the IPFwd application.

POT Peak Over Threshold.

PPS Packets Per Second.

R Receiving thread of the IPFwd application.

SGMS Stream Graph Modulo Scheduling.

SMT Simultaneous Multithreading.

STAP Space Time Adaptive Processing.

T Transmitting thread of the IPFwd application.

TA Thread Assignment.

TCP Transmission Control Protocol.

TLB Translation Lookaside Buffer.

TLP Thread Level Parallelism.

TSBSched Thread-to-Strand Binding Scheduler.

UD Uniformly Distributed.

UDP User Datagram Protocol.

UPB Upper Performance Bound.

vCPU Virtual CPU (Central Processing Unit).

WCET Worst-Case Execution Time.