Thread Assignment of Multithreaded Network Applications in Multicore/Multithreaded Processors

Petar Radojković, Vladimir Čakarević, Javier Verdú, Alex Pajuelo,
Francisco J. Cazorla, Mario Nemirovsky and Mateo Valero

Abstract—The introduction of multithreaded processors comprised of a large number of cores with many shared resources makes thread scheduling, and in particular optimal assignment of running threads to processor hardware contexts to become one of the most promising ways to improve the system performance. However, finding optimal thread assignments for workloads running in state-of-the-art multicore/multithreaded processors is an NP-complete problem.
In this paper, we propose *BlackBox scheduler*, a systematic method for thread assignment of multithreaded network applications running on multicore/multithreaded processors. The method requires minimum information about the target processor architecture and no data about the hardware requirements of the applications under study. The proposed method is evaluated with an industrial case study for a set of multithreaded network applications running on the UltraSPARC T2 processor. In most of the experiments, the proposed thread assignment method detected the best actual thread assignment in the evaluation sample. The method improved the system performance from 5% to 48% with respect to load balancing algorithms used in state-of-the-art OSs, and up to 60% with respect to a naive thread assignment.

Index Terms—Chip Multithreading (CMT), Process scheduling, Performance modeling

1 Introduction

Lightweight Kernel (LWK) implementations strive to provide applications with predictable performance and maximum access to the hardware resources of the system. In order to achieve these goals, Operating System (OS) services are restricted to only those that are absolutely necessary. Furthermore, the provided services are streamlined, reducing to the minimum the overhead of the LWK. Also, LWKs usually apply simplified algorithms for thread scheduling and memory management that provide a significant and predictable amount of the processor resources to the running applications.

Dynamic scheduling may vary the amount of processing time made available to applications during their execution, which can significantly affect the performance of High Performance Computing (HPC) applications [19][29], and reduce the performance provided by commercial network processors. As a result, many systems already use LWKs with static scheduling, such as CNK [36] in BlueGene HPC systems, and Netra DPS [2][3] mainly used in network environments.

Multithreaded processors [1] support concurrent execution of several threads which improves the utilization of hardware resources and the overall system performance. On the other hand, the execution of several threads at a time significantly increases the complexity of thread scheduling. As concurrently running threads (co-runners) interfere in processor resources, system performance significantly depends on the characteristics of the co-runners and their distribution on the processor.

Multithreaded processors that comprise several cores, where each core supports several concurrently running threads, have different levels of resource sharing [41]. For example, in a CMP processor where each core supports concurrent execution of several threads through SMT, all co-running threads share global resources such as the last level of cache or the I/O. In addition to this, threads running in the same core share core resources such as the instruction fetch unit, or the L1 instruction and data cache. Therefore, the way that threads are assigned to the cores determines which resources they share, which may significantly affect system performance (see Section 2.2 of the supplementary file). In processors with several levels of resource sharing, thread scheduling comprises two steps. In the first step, usually called *workload selection*, from all ready-to-run threads, the OS selects the set of threads (workload) that will be executed on the processor in the next time slice. In the second step, called *thread assignment*, each thread in the workload is assigned to a hardware context (virtual CPU) of the processor.

In this paper, we propose *BlackBox scheduler*, a systematic method for a thread assignment of multithreaded network applications 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 profiling thread assignments that can be used to model the interference between concurrently running threads. 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. BlackBox scheduler enhances the TSBSched [31] that is, to the best of our knowledge, the first scheduler that addresses the problem of thread assignment of multithreaded applications. As we explain in Section 2.4 of the supplementary file, the main limitation of TSBSched is that it requires significant knowledge about the application source code and changes of the code. If the application source code is

- *P. Radojković and V. Čakarević are with Barcelona Supercomputing Center (BSC), Barcelona, Spain. email: {petar.radojkovic, vladimir.cakarevic}@bsc.es*
- *J. Verdú and A. Pajuelo are with Universitat Politècnica de Catalunya (UPC), Barcelona, Spain. email: {jverdu, mpajuelo}@ac.upc.edu*
- *F. Cazorla is Scientific Researcher in the Spanish National Research Council (IIIA-CSIC) and with BSC, Barcelona, Spain. email: francisco.cazorla@bsc.es*
- *M. Nemirovsky is ICREA Research Professor and with BSC, Barcelona, Spain. email: mario.nemirovsky@bsc.es*
- *M. Valero is with UPC and BSC, Barcelona, Spain. email: mateo@ac.upc.edu*

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

not available, TSBSched cannot be used. BlackBox scheduler is designed to overcome these limitations.

The proposed thread assignment method is evaluated with an industrial case study for a set of multithreaded networking applications running on the UltraSPARC T2 processor. In most of the experiments, BlackBox scheduler detected the best actual (measured) thread assignment. The highest performance difference between the thread assignment provided by the method and the actual best thread assignments in the evaluation sample is only 1.4%. BlackBox scheduler also provides a significant performance improvement with respect to the state-of-the-art thread assignment techniques.

The rest of the paper is organized as follows. Section 2 introduces multithreaded processors with several levels of resorce sharing and describes the UltraSPARC T2 processor used in the study. The details of BlackBox scheduler are described in Section 3. Section 4 describes the experimental environment used in the study. The results of the experiments used in the evaluation of BlackBox scheduler are presented in Section 5. The related work is presented in Section 6, while Section 7 lists the conclusions of the study.

2 MULTITHREADED PROCESSORS WITH SEVERAL LEVELS OF RESOURCE SHARING

In this paper, we focus on the problem of thread assignment of multithreaded network applications that are running on multithreaded processors with several levels of resource sharing.

Multithreaded processors that comprise several cores where each core supports concurrent execution of several threads, have different levels of resource sharing. For example, in a CMP processor where each core can execute several threads at a time through multithreading, co-running threads share and compete for globally-shared resources such as the last level of cache or the I/O. In addition to this, the threads running on the same core also share the resources that are private to each core, such as the integer and floating point unit, or the L1 instruction and data cache. Therefore, the way that the threads are assigned to the processor cores determines which resources they share, which may significantly affect the system performance.

2.1 The UltraSPARC T2 processor

A good example of the multithreaded processor with several levels of resource sharing is the UltraSPARC T2 processor. The UltraSPARC T2 [1] is a multithreaded processor that comprises eight cores connected through the crossbar to the shared L2 cache (see Figure 1). Each of the cores includes eight hardware contexts for a total of 64 software threads being executed concurrently. Hardware contexts inside the core are divided into two groups of four contexts, forming two hardware execution pipelines. Thus, threads that concurrently execute on the UltraSPARC T2 processor share (and compete for) different resources in three different levels depending on how they are distributed on the processor (see Figure 1).

The resources at the ***InterCore*** level are shared between all threads concurrently executing on the processor [41]. Resources shared at this level are mainly the L2 cache, the on-chip interconnection network (crossbar), the memory controllers, and the interface to off-chip resources such as the I/O.

In addition to resources shared at InterCore level, threads running in the same core share ***IntraCore*** processor resources: the L1 instruction and data cache, the instruction and data TLBs, the Load Store Unit (LSU), the Floating Point and Graphic Unit (FPU), and the Cryptographic Processing Unit.

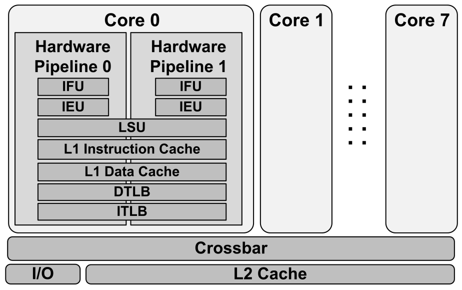


Fig. 1. Schematic view of the three resource sharing levels of the UltraSPARC T2 processor

Finally, the threads that execute in the same hardware pipeline also share ***IntraPipe*** resources: the Instruction Fetch Unit (IFU) and the Integer Execution Units (IEU).

In order to fully utilize the performance of the multithreaded processors like the UltraSPARC T2, it is important to understand which hardware resources are shared on each resource-sharing level, and to distribute the concurrently running threads in such a way that the collision in the shared resources is minimized. On the other hand, the sharing of the hardware resources can also improve the performance. Threads running in the same processor core communicate through the shared L1 cache, while threads running in different cores share data and instructions only at globally shared L2 cache which has significantly higher access time. Therefore, the threads that share instructions or data should be co-scheduled in the same L1 cache domains (the same cores in the case of the UltraSPARC T2 processor) in order to improve the code reuse and reduce the latency of the inter-thread communication.

3 BLACKBOX SCHEDULER

We present a systematic method for thread assignment of multithreaded network applications running on processors with several levels of resource sharing. The purpose of this method 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 method determines a set of thread assignments (profiling thread assignments) that are used to model the interference 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.

The proposed thread assignment method does not require the information about the execution time of each application thread nor does it analyze the slowdown experienced by each thread independently. When the method analyzes the slowdown due to inter-thread interferences, the application is seen as a black box. Therefore, we refer to the proposed method as *BlackBox scheduler*. BlackBox scheduler is 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 6) 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 method requires 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 method does 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 this method to different processor architectures requires minimum adjustments.

3.1 The algorithm

Figure 2 presents the schematic view of BlackBox scheduler. The method is comprised of three phases: (1) Application profiling, (2) Performance prediction, and (3) Selection phase.

(1) **Application profiling phase:** We profile the set of applications that we want to schedule. 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 of the applications under study.

(2) **Performance prediction phase:** The model 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 provided performance.

The following sections describe each phase of the thread assignment method in detail.

3.1.1 Phase 1: Application profiling

Application profiling is the first phase of BlackBox scheduler. In this phase, we measure the interference between concurrently running threads for 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 [41] as a representative of processors with several levels of resource sharing.

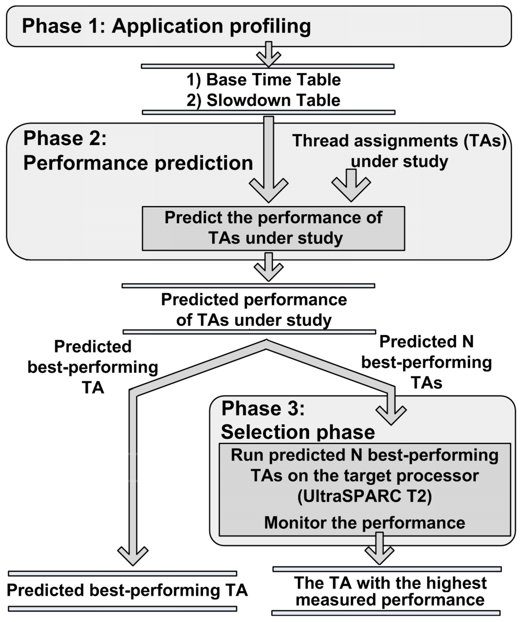

Phase 1: Application profiling
1) Base Time Table
2) Slowdown Table
Phase 2:
Performance prediction
Thread assignments (TAs) under study
Predict the performance of TAs under study
Predicted performance of TAs under study
Predicted best-performing TA
Predicted N best-performing TAs
Phase 3:
Selection phase
Run predicted N best-performing TAs on the target processor (UltraSPARC T2)
Monitor the performance
Predicted best-performing TA
The TA with the highest measured performance

Fig. 2. Schematic view of the thread assignment method

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 BlackBox scheduler. Instead of the analyzing 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. Based on this conclusion, when we model the interference between co-running threads, we focus on the interaction between threads running on the same core, i.e. we disregard the threads running on the remote cores.

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 combinations of the workload inside the processor core. We measure the performance of the target thread in each experiment, and store this data into a table. As 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 4.3). 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 it for 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$_{tg}$-P$_{tg}$-T$_{tg}$ application experiences when R$_{tg}$ 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 combinations 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 performance of the *target application* R$_{tg}$-P$_{tg}$-T$_{tg}$ in the given thread assignment. In all the experiments for R$_{tg}$ thread, P$_{tg}$ and T$_{tg}$ threads execute in isolation on remote cores. This way we are sure that the cause of the observed performance variation is the interference between R$_{tg}$ and stressing threads, and not any slowdown of the threads P$_{tg}$ and T$_{tg}$. 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 UltraSPARC T2 core 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 application performance in each of them. 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 application performance in a given assignment.

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) run on the same processor core, they communicate through L1 data cache. When the threads run 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 application performance in each assignment. The set of experiments is presented in Figure 3. 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. Therefore, the comparison of the performance of the application in *TA7* and *TA11* can provide the information about the benefit of data and instruction sharing between R and T threads.

3.1.2 Phase 2: Performance prediction

Performance prediction is the second phase of the thread assignment method, see Figure 2. In this phase, based on the profiling data, the Base Time Table and Slowdown Table, BlackBox scheduler 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:

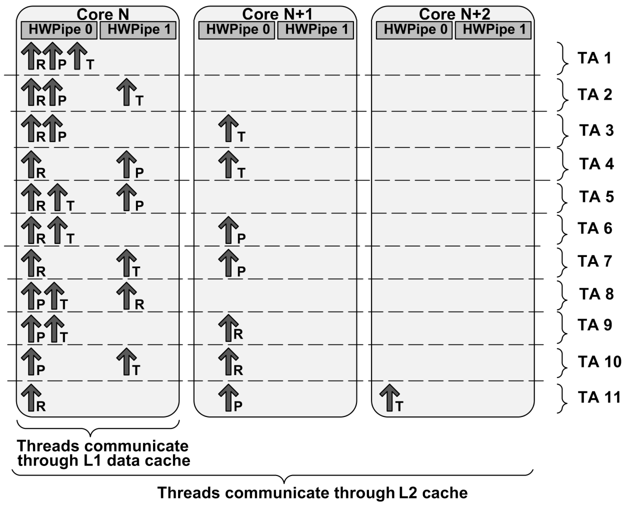


Fig. 3. Experiments for the Base Time Table

In *Step 1*, we analyze each application in the workload independently, as it executes in isolation. Hence, we disregard the interference between different applications running on the processor, and model only the interaction between threads that belong to the same application. In this step, each application in the workload is associated with its *base_performance*, the performance that the application would have as if it was 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 BlackBox scheduler with an example thread assignment that is presented in Figure 4. 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 on 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 R1-P1-T1 application we disregard R2, P2, and T2 threads from the original thread assignment, and vice versa (see Figure 4). In this step, each application is associated with its *base_performance*, the performance that the application would have if it was executed alone on the processor. The *base_performance* is directly read from the Base Time Table (see Section 3.1.1).

In *Step 1.1*, we analyze R1-P1-T1 application in the example thread assignment presented in Figure 4. We directly read the *base_performance* of R1-P1-T1 application from the field of

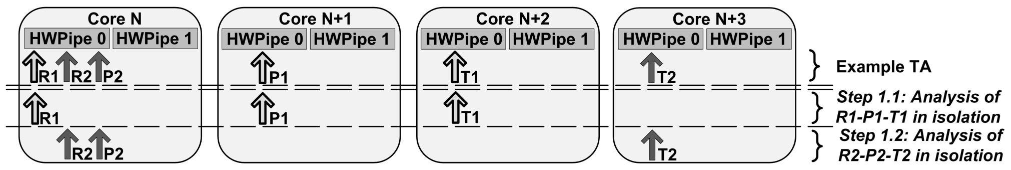

Core N
Core N+1
Core N+2
Core N+3
HWPipe 0
HWPipe 1
R1 R2 P2
P1
T1
T2
Example TA
Step 1.1: Analysis of R1-P1-T1 in isolation
Step 1.2: Analysis of R2-P2-T2 in isolation

Fig. 4. Performance prediction. Step 1: Analysis of each application instance in isolation.

the Base Time Table that corresponds to the thread assignment 11 (*TA11*) presented in Figure 3 (R, P, and T threads assigned to different cores). The *base_performance* of R2-P2-T2 application is computed analogously. First we observe R2-P2-T2 application as if it was executed in isolation (Step 1.2 of Figure 4). Then we read the corresponding field of the Base Time Table, the field that corresponds to *TA3* presented in Figure 3.

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 3.1.1, 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 5 presents the part of the example thread assignment that we use to illustrate the *Step 2* of the performance prediction. The part of the thread assignment that we analyze comprises three threads, R1, R2, and P2, that execute on the same hardware pipeline of core N. Figure 5 present also the entries of the Slowdown Table that are used in the analysis.

In *Step 2.1* presented in Figure 5, we compute *slowdown(R1)*, the slowdown that the application R1-P1-T1 experiences because of the interference of thread R1 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. As R1 thread is the thread under analysis, it corresponds to the R_{tg} in the Slowdown Table. We want to detect the slowdown that threads R2 and P2 cause to R1-P1-T1 application. Thus, threads R2 and P2 correspond to P_{st} and T_{st} threads in the Slowdown Table. First, from the Slowdown Table, we read the performance of the target application R_{tg}-P_{tg}-T_{tg} when thread R_{tg} executes alone on the processor core (*performance 1*). This entry corresponds to the thread assignment in which thread R1 executes in isolation on the processor core. Later, we read the performance of the application R_{tg}-P_{tg}-T_{tg} when thread R_{tg} runs on the same hardware pipeline with one R_{st} and one P_{st} thread (*performance 2*). This corresponds to thread R1 running with threads R2 and P2 on the same hardware pipeline in a processor core. Finally, the slowdown that the application R1-P1-T1 experiences because of the interference of thread R1 with threads R2 and P2 is computed as the ratio between *performance 1* and *performance 2*, *slowdown(R1)*=$\frac{performance\ 1}{performance\ 2}$. In all the experiments, threads P_{tg} and T_{tg} are executed in isolation on remote cores.

Step 2.2 determines the slowdown that the application R2-P2-T2 experiences because thread R2 interferes with the threads running in the same core (*slowdown(R2)*). In *Step 2.3*, we compute *slowdown(P2)*, the slowdown that application R2-P2-T2 experiences because thread P2 collides in processor core resources with threads R1 and R2. As we show in Figure 5, *Step 2.2* and *Step 2.3* are analogous to *Step 2.1*. Threads P1, T1, and T2 of the example thread assignment

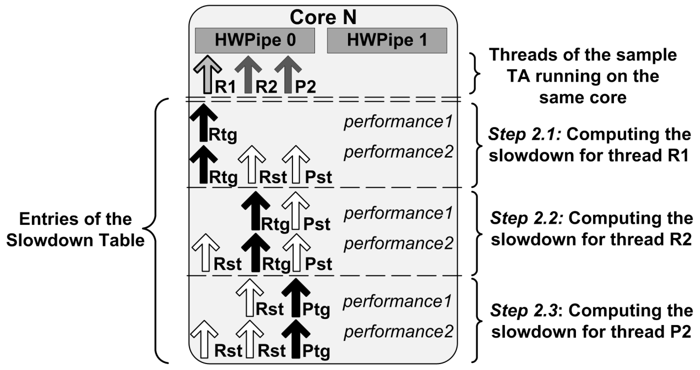

Core N
HWPipe 0
HWPipe 1
R1 R2 P2
Threads of the sample TA running on the same core
Rtg
Rtg Rst Pst
performance1
performance2
Step 2.1: Computing the slowdown for thread R1
Rtg Pst
Rst Rtg Pst
Step 2.2: Computing the slowdown for thread R2
Rst Ptg
Rst Rst Ptg
Step 2.3: Computing the slowdown for thread P2
Entries of the Slowdown Table

Fig. 5. Performance prediction. Step 2: Modeling the collision in hardware resources.

execute alone on different processor cores (see Figure 4), 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 performance of a given thread assignment. First, we read the *base_performance* of each application in a given thread assignment (see *Step 1*). Later, as we explained in *Step 2*, we compute the slowdown that the application experiences because the threads it comprises interfere in shared processor resources with co-running applications. The slowdown that a multithreaded application experiences because of the collision in hardware resources corresponds to the highest slowdown of each thread independently. For example, the slowdown that the application R1-P1-T1 experiences is computed as: *slowdown(R1-P1-T1)* =
= MAX[*slowdown(R1), slowdown(P1), slowdown(T1)*].

The performance of each application is computed as a ratio between the *base_performance* and *slowdown* of the application. For example: *performance(R1-P1-T1)* = $\frac{base\_performance(R1-P1-T1)}{slowdown(R1-P1-T1)}$. The $base\_performance$(R1-P1-T1) corresponds to the performance that the application would have had if it was executed alone on the processor. Factor $slowdown$(R1-P1-T1) quantifies the the impact on R1-P1-T1 performance because of collision with co-running threads. Finally, the performance of a given thread assignment is a sum of performance of all the applications that it comprises.

In the Prediction phase, we predict the performance of thousands of different thread assignments. The output of BlackBox scheduler can be the thread assignment with the highest predicted performance, or, optionally, the prediction can be improved in the Selection phase.

3.1.3 Phase 3: Selection phase

Selection phase is the final phase of BlackBox scheduler, see Figure 2. Although BlackBox scheduler predicts the thread assignment performance with a high accuracy (as we show in Section 5), 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 BlackBox scheduler 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 2nd, 3rd, or Nth predicted best-performing assignment. In Section 5, we analyze also the impact of the Selection phase to the performance of BlackBox 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 8%, which is significant.

3.2 Scalability

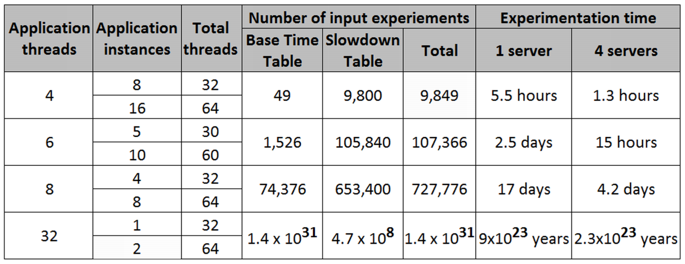
The data used to predict the performance of thread assignments is stored in the Base Time Table and the Slowdown Table. In order to collect data for the Base Time Table, we have to execute all thread assignments of a single application on the target processor. The Slowdown Table requires running all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

Since state-of-the-art networking applications comprise few threads [3][42], the execution of experiments needed to fill the Base Time Table and the Slowdown Table is feasible, and BlackBox scheduler 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. Complex network applications, like network security, usually comprise up to three threads (Receiving-Processing-Transmitting), due to complexity of splitting the processing among different threads in an optimal way.

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 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 BlackBox scheduler can be used if the number of application threads increases. The first column of Table 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 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*

TABLE 1
Scalability of the proposed thread assignment method



Application threads	Application instances	Total threads	Number of input experiments			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					

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 method 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, BlackBox scheduler also has 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 3.1.1). 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 combinations of the workload inside the processor core.

Table 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.

As a part of future work, we plan to enhance the application profiling and the performance prediction algorithm, so BlackBox scheduler requires less profiling experiments. Our goal is to reduce the time needed to execute the application profiling experiments and to make BlackBox scheduler applicable for applications that comprise large number of threads running on processors with large number of cores and hardware contexts per core.

4 Experimental environment

We evaluate BlackBox scheduler on a set of real multithreaded network applications running on the UltraSPARC T2 processor. In order to avoid interferences between user applications and operating system processes, we execute our experiments in Netra DPS, a low-overhead environment used in network processing systems [2][3]. We briefly describe Netra DPS being focused on the differences between this environment and fully-fledged operating systems, such as Linux or Solaris. At the end, we present the set of multithreaded network applications and the methodology we use to evaluate the BlackBox scheduler.

4.1 Hardware environment

In order to evaluate BlackBox scheduler, we use an industrial experimental environment used in network systems. The environment comprises two SPARC Enterprise T5220 servers [4] that manage the generation and the processing of the network traffic. Each T5220 server comprises one UltraSPARC T2 processor and 64GB of the memory. One server executes the Network Traffic Generator (NTGen) [3]. NTGen is a software tool, developed by Oracle that generates IPv4 TCP/UDP packets with configurable various packet header fields. NTGen transmits the network packets through the 10Gb link to the second T5220 server in which we execute the benchmarks under study. NTGen generates enough traffic to saturate the network processing machine in all experiments presented in the study. Thus, in all experiments the performance bottleneck is the speed at which packets are processed, which is determined by the performance of the selected thread assignments.

4.2 Netra DPS

Networking systems use lightweight runtime environment to reduce the overhead introduced by fully-fledged OSs. One of these environments is Netra DPS [2][3] developed by Oracle.

Netra DPS does not incorporate virtual memory nor runtime process scheduler, and performs no context switching. The assignment of running tasks to processor hardware contexts (virtual CPUs) is performed statically at the compile time. It is the responsibility of the programmer to define the hardware context in which each particular task will be executed. Netra DPS does not provide any interrupt handler nor daemons. A given tasks runs to completion on the assigned hardware context without any interruption.

4.3 Benchmarks

Netra DPS is a 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 cannot be executed in this environment without previous benchmark adjustments. The benchmarks we use in the study are designed based on the IP Forwarding application (IPFwd) that is included in the Netra DPS distribution [2]. IPFwd is one of the most representative low-layer network applications. The application reads a packet header and makes the decision where to forward the packet based on the destination IP address. The IPFwd application that we use in the experiments consists of three threads that form a software pipeline (see Figure 6):

- The receiving thread (R) reads a packet from the Network Interface Unit (NIU) and writes the pointer to the packet into the R$\rightarrow$P memory queue.
- The processing thread (P) reads the pointer to the packet from the memory queue, processes the packet by hashing on the lookup table, and writes the pointer to the P$\rightarrow$T memory queue.
- Finally, the transmitting thread (T) reads the pointer from the P$\rightarrow$T memory queue and transmits the packet to the network through the NIU.

The default IPFwd comprises a simple processing stage. However, current and future network services include other applications, like Deep Packet Inspection and Intrusion Detection that perform complex processing of the packets. Complex packet processing increases the number of instructions that are executed during the packet processing and increases the working data set. In addition to this, depending on the features of network applications and the network traffic, the applications can have significantly different memory behavior.

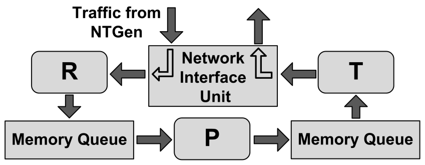


Fig. 6. One instance of the IPFwd application

In order to mimic a variety of network applications we create nine variants of the IPFwd application in which we explore different memory behavior and different complexity of the processing thread. We start by adjusting the IPFwd lookup table to cover from the best to the worst case in terms of cache utilization: (1) The lookup table fits in the L1 data cache (*hash1_L1*); (2) The table does not fit in the L1 data cache, but it fits in the L2 cache (*hash1_L2*); (3) The lookup table entries are initialized to make IPFwd continuously access the main memory (*hash1_Mem*) that is representative of the worst case assumptions used in network processing studies [35].

In order to emulate more complex packet processing and multiple accesses to memory (e.g. Deep Packet Inspection), we repeat the hash function call and the hash table lookup N times (three times in experiments presented in this paper). We refer to this application as *hashN*. Again, we use three different configurations to analyze complementary scenarios of cache utilization: *hashN_L1*, *hashN_L2*, and *hashN_Mem*.

Finally, in order to mimic CPU-intensive network applications, such as high-layer packet decoding or URL decoding we design *intadd*, *intmul*, and *intdiv* benchmarks. *intadd*, *intmul*, and *intdiv* benchmarks are developed by inserting a set of integer addition, multiplication, and division instructions, respectively, at the end of the IPFwd processing stage. These benchmarks put high stress to IntraPipe and IntraCore processor resources.

The benchmarks that we use in the study perform different kinds of processing and stress different processor resources. They have different duration of processing threads and different bottleneck threads. The presented benchmarks stress the UltraSPARC T2 hardware resources at all three sharing levels, and stress the most critical hardware resources at each level: the Instruction Fetch Unit at IntraPipe, the non-pipelined execution units at IntraCore, and the L2 cache at InterCore level [41]. Considering all this, we argue that these benchmarks represents a good testbed for the BlackBox scheduler evaluation.

4.4 Baseline

Since the number of possible thread assignments is vast [13][14][20][31], it is unfeasible to do the exhaustive search in order to find the thread assignment with the highest performance. Also, the analytical analysis of the optimal thread assignment is an NP-complete problem [18].

Previous studies that address thread assignment problem [5][14][31] verify their proposals with respect to a naive thread assignment in which the threads are randomly assigned to the virtual CPUs of the processor, or with respect to balanced thread assignments. In balanced assignments the threads are equally distributed among processor hardware domains. State-of-the-art scheduling algorithms used in Linux and Solaris [7][32] intend to equally distribute the running threads among different hardware domains (for UltraSPARC T2 processor, among processor cores and hardware pipelines). The purpose of this approach is to distribute running threads to equally stress the hardware resources of the processor. Since the current Linux and Solaris load balancing algorithms disregard inter-thread dependencies and different resource requirements of each thread, the balanced thread assignments exhibit non-optimal performance in general case.

In addition to the naive and balanced thread assignment, we compare the performance of the method presented in this paper with the performance provided by *TSBSched* (Thread-to-Strand Binding Scheduler)[31], which is, to the best of our knowledge, the only systematic method for thread assignment of multithreaded applications. State-of-the-art thread assignment approaches are described in Section 2.3 of the supplementary file.

4.5 Methodology

In order to assure that the measurements are taken while the system was in the steady state, we measured the performance of thread assignments when each application instance processed from three to four million packets. This means that each application thread was executed from three to four million times. The duration of each experiment is at least 1.5 seconds, and it depends on the benchmark, the number of the application instances, and the distribution of the concurrently running threads. For each experiment, we computed the mean value and the standard deviation of the measured performance. In all the experiments presented in the paper, the standard deviation is negligible, thus we do not report it in the presented charts. This is expected because the experiments were executed in Netra DPS, a low-overhead runtime environment that is designed to provide a stable execution time [30].

5 Evaluation

In this section, we evaluate BlackBox scheduler for the benchmarks presented in Section 4.3. First, we execute several application instances in different thread assignments, and measure the performance of each assignment. Then, we use BlackBox scheduler to predict the performance of these assignments. In the next step, we compare the performance of the predicted best-performing assignment with the performance of actual best one that is captured in the evaluation sample. Finally, we present the performance improvement of our method over a naive scheduling, load balancing scheduling algorithms used in the state-of-the-art OSs, and TSBSched. As a main metric, we use the number of processed Packets Per Second (PPS). This metric is inversely proportional to the packet processing time, and has the same properties as the execution time for non-packet oriented applications. We present results for 6 and 24 concurrently running application threads. The results for 9, 12, and 18 concurrently running threads are presented in Section 3 of the supplementary file.

5.1 Six application threads

As the total number of different thread assignments for six co-running threads is around 1500, we were able to generate, execute, and measure the performance of all of them. The results are presented in Figure 7. Different benchmarks are listed along the X-axis of the figure, while the Y-axis shows the slowdown with respect to the best actual (measured) thread assignment. We present several groups of bars. Bars *BlackBox* show the performance difference between the actual best

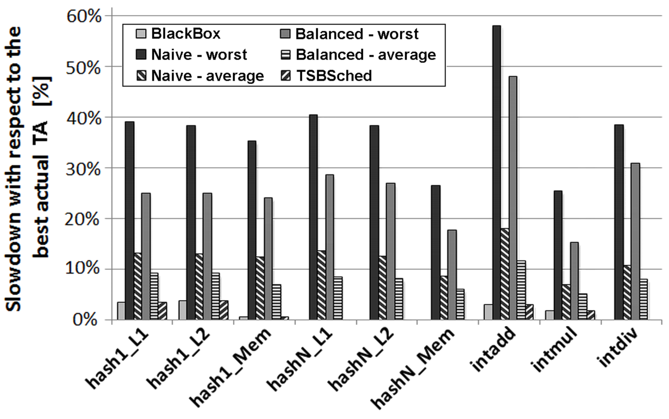


Fig. 7. Slowdown with respect to the best actual thread assignment (six application threads)

thread assignment and the assignment provided by BlackBox scheduler. In these experiments, the selection phase is not included, and the thread assignment with the predicted highest performance is the final outcome of BlackBox scheduler. Bars *Naive-average* and *Naive-worst* present the average and the worst performance loss of naive scheduling with respect to the actual best thread assignment. Bars *Balanced-average* and *Balanced-worst* present the average and the worst slowdown of balanced thread assignments. Finally, *TSBSched* bars present the performance of TSBSched thread assignment approach.

For six concurrently running threads, BlackBox scheduler predicts the actual best thread assignment for four out of nine benchmarks: *hashN_L1*, *hashN_L2*, *hashN_Mem*, and *intdiv*. The performance difference between the predicted best-performing thread assignment and the actual best one is always below 4%. The improvement of BlackBox scheduler with respect to naive and balanced scheduling techniques is significant. The performance improvement with respect to a naive process scheduling is between 5% and 15% in average, and it ranges up to 55% (*intadd*). The performance improvement with respect to balanced thread assignments is between 3% and 9% in average, and it ranges up to 45% (*intadd* benchmark). Finally, in the experiments for six application threads, BlackBox scheduler has precisely the same accuracy as the TSBSched approach for all nine benchmarks in the suite.

The main goal of BlackBox scheduler is not to improve the performance of TSBSched approach. The performance provided by TSBSched is already close to the optimal one, thus it cannot be significantly improved [31]. The main enhancement of BlackBox scheduler with respect to TSBSched is that our method does not require the understanding and adjustments of the application source code. Unlike TSBSched, BlackBox scheduler can be used when application source code is unavailable, which is a common case in real environments.

Figure 7 presents the results when BlackBox scheduler does not include the selection phase, but the final outcome is the best predicted thread assignment. Figure 8 shows the performance improvement of the selection phase. The X-axis of the figure shows the number of thread assignment executed in the selection phase, while the Y-axis shows the slowdown with respect to the best actual assignment. We show results for all nine benchmarks in the suite. Presented results show that the selection phase additionally improves the performance for four out of nine benchmarks in the suite. In four out of remaining five benchmarks, the best predicted thread assignment is also the best actual one, so the performance cannot be improved because it is already the highest possible.

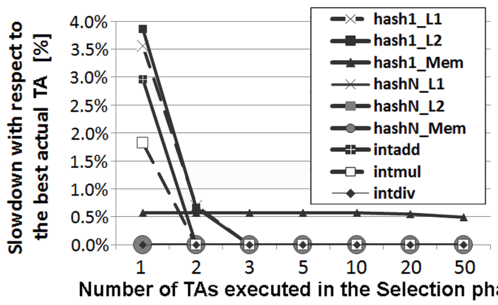


Fig. 8. The performance improvement of the selection phase (six application threads)

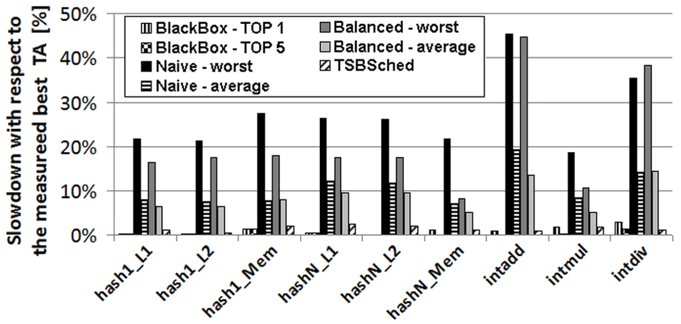


Fig. 9. Slowdown with respect to the measured best thread assignment (24 application threads)

Performance improvement of the selection phase ranges from 1.8% (*intmul*) to 3.9% (*hash1_L2*).

5.2 24 application threads

The results for 24 concurrently running application threads are presented in Figure 9. Different benchmarks are listed along the X-axis of the figure, while the Y-axis shows the slowdown with respect to the best measured thread assignment in the evaluation sample. We present several groups of bars. Bar *BlackBox - TOP1* shows the performance difference between actual best thread assignment in the sample and the one predicted by BlackBox scheduler without the selection phase. Bar *BlackBox - TOP5* shows the performance difference when five predicted best-performing assignments are executed in the selection phase of BlackBox scheduler. The rest of the bars is the same as in Figure 7. The number of possible thread assignments for 24 co-running threads is vast, and it is unfeasible to generate and to predict the performance for each assignment. Therefore, the proposed thread assignment method is evaluated on a sample 1000 random assignments, which is a correct statistical approach. The method that generates random thread assignments is described in detail in Section 3.1.1 of the supplementary file.

When 24 application threads concurrently execute on the processor (see Figure 9) the highest performance loss introduced by the method without the selection phase is only 2.8% (*intdiv* benchmark). When the selection phase is included, the highest performance loss is only 1.4% (*hash1_Mem*).

As in experiments for six application threads, BlackBox scheduler introduces significant performance improvement with respect to a naive scheduling and load balancing algorithms. The improvement with respect to a naive scheduling is between 6% to 19% in average, and up to 45% in the worst case (*intadd* benchmark). The performance improvement with respect to load balancing scheduling techniques ranges from 6% (*hash1_Mem*) to 19% (*intadd*) in average, and up to 45% in the worst case (*intadd* benchmark).

We could not execute more than 24 benchmark threads 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 thread assignment method to applications with several processing threads and to workloads with a higher number of concurrently running threads.

6 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. [37][38] present the SOS scheduler, the approach that uses hardware performance counters to find schedules that exhibit good performance. Eyerman and Eeckhout [16] propose probabilistic job symbiosis model that enhances the SOS scheduler. Based on the cycle accounting architecture [15][26][27], the model estimates the single-threaded progress for each job in a multithreaded workload. Other approaches [12][17][21][33] propose techniques to construct workloads of threads that exhibit good symbiosis in shared caches solving problems of cache contention.

Kwok and Asmad [24][25] 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 [6][10][40] and real-time systems [8][9][11]. 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 [5][14][31]. Acosta et al. [5] 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. [14] 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. To the best of our knowledge, the only study that addresses the problem of thread assignment of multithreaded applications is our previous work in which we present TSBSched [31]. TSBSched is a model that predicts the performance of multithreaded network applications running on processors with several levels of resource sharing. TSBSched and its main limitations are described in Section 2.4 of the supplementary file.

Other studies analyze thread scheduling for platforms comprised of several multithreaded processors [28][39]. McGregor et al. [28] 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. Tam el al. [39] present a run-time technique for the detection of data sharing among different threads. The proposed technique can be used by an operating system job scheduler to assign threads that share data to the same memory domain (same chip or the same core on the chip).

Kumar et al. [23] and Shelepov et al. [34] 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 among 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. [22] propose an algorithm that assigns network processing tasks to processor cores with the goal of reducing the power consumption. Wolf et al. [43] 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 BlackBox scheduler, a thread assignment method for network applications running on multithreaded processors with several level of resource sharing. To the best of our knowledge, BlackBox scheduler is the first approach that predicts the performance of multithreded applications in different thread assignments without profound understanding and adjustments of the application source code. The method profiles applications by measuring the overall application performance in different thread assignments, and does not require inserting any test probes in the application code. This is very important because BlackBox scheduler can be applied if the application source code or parts of it are unavailable, which is a common case in the industry.

7 Conclusions

Optimal thread assignment is one of the most promising ways to improve performance in LWK environments running on multithreaded processors. However, finding an optimal thread assignment on modern multithreaded processors comprising a large number of cores is an NP-complete problem.

In this paper, we presented BlackBox scheduler, a method for systematic thread assignment of multithreaded network applications running on processors with several levels of resource sharing. BlackBox scheduler is evaluated for a set of multithreaded networking applications running on the UltraSPARC T2 processor. The presented results show a high accuracy of the method that determined the best thread assignments in the evaluation sample in most of the experiments.

BlackBox scheduler demonstrated significant performance improvement over the naive scheduling and schedulers that apply load balancing. The method improves the performance of naive scheduling from 6% to 23% in average, and up to 60% in the worst case. The performance improvement with respect to load balancing schedulers ranges from 5% to 22% in average, and up to 48% in the worst case.

References

[1] *OpenSPARC^{TM} T2 Core Microarchitecture Specification*. Sun Microsystems, Inc, 2007.
[2] *Netra Data Plane Software Suite 2.0 Update 2 Reference Manual*. Sun Microsystems, Inc, 2008.
[3] *Netra Data Plane Software Suite 2.0 Update 2 User's Guide*. Sun Microsystems, Inc, 2008.
[4] *Oracle data sheet: Sun SPARC Enterprise T5220 Server*. Oracle, 2009.
[5] C. Acosta *et al.*, "Thread to Core Assignment in SMT On-Chip Multiprocessors," in *SBAC-PAD 2009*.
[6] C. Boneti *et al.*, "Software-controlled priority characterization of power5 processor," in *ISCA 2008*.
[7] D. Bovet and M. Cesati, *Understanding the Linux Kernel*. O'Reilly Media, Inc., 2006.
[8] F. J. Cazorla *et al.*, "Architectural support for real-time task scheduling in smt processors," in *CASES 2005*, ser. CASES '05.
[9] F. J. Cazorla *et al.*, "Predictable performance in smt processors: Synergy between the os and smts," *IEEE Trans. Comput.*, vol. 55, no. 7, 2006.
[10] F. J. Cazorla *et al.*, "Dynamically controlled resource allocation in smt processors," in *MICRO 2004*.
[11] F. J. Cazorla *et al.*, "Qos for high-performance smt processors in embedded systems," *IEEE Micro*, vol. 24, no. 4, 2004.
[12] D. Chandra *et al.*, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *HPCA 2005*.
[13] M. De Vuyst *et al.*, "Exploiting unbalanced thread scheduling for energy and performance on a CMP of SMT processors," in *IPDPS 2006*.
[14] A. El-Moursy *et al.*, "Compatible phase co-scheduling on a CMP of multi-threaded processors," in *IPDPS 2006*.
[15] S. Eyerman and L. Eeckhout, "Per-thread cycle accounting in SMT processors," in *ASPLOS 2009*.
[16] S. Eyerman and L. Eeckhout, "Probabilistic job symbiosis modeling for SMT processor scheduling," in *ASPLOS 2010*.
[17] A. Fedorova *et al.*, "Performance of multithreaded chip multiprocessors and implications for operating system design," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2005.
[18] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.
[19] R. Gioiosa *et al.*, "Analysis of system overhead on parallel computers," in *4th IEEE International Symposium on Signal Processing and Information Technology*, 2004.
[20] Y. Jiang *et al.*, "Analysis and approximation of optimal co-scheduling on chip multiprocessors," in *PACT 2008*.
[21] J. Kihm *et al.*, "Understanding the impact of inter-thread cache interference on ILP in modern SMT processors," *The Journal of Instruction Level Parallelism*, vol. 7, 2005.
[22] R. Kokku *et al.*, "A case for run-time adaptation in packet processing systems," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 1, 2004.
[23] R. Kumar *et al.*, "Single-ISA heterogenous multi-core architectures for multithreaded workload performance," in *ISCA 2004*.
[24] Y.-K. Kwok and I. Ahmad, "Benchmarking and comparison of the task graph scheduling algorithms," *Journal of Parallel and Distributed Computing*, vol. 59, no. 3, Dec. 1999.
[25] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Computing Surveys*, vol. 31, no. 4, Dec. 1999.
[26] C. Luque *et al.*, "ITCA: Inter-task Conflict-Aware CPU Accounting for CMPs," in *PACT 2009*.
[27] C. Luque *et al.*, "CPU Accounting in CMP Processors," *in IEEE Computer Architecture Letters*, vol. 8, no. 1, 2009.
[28] R. L. McGregor *et al.*, "Scheduling algorithms for effective thread pairing on hybrid multiprocessors," in *IPDPS 2005*.
[29] F. Petrini *et al.*, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q," in *SC 2003*.
[30] P. Radojković *et al.*, "Measuring Operating System Overhead on CMT Processors," in *SBAC-PAD 2008*.
[31] P. Radojković *et al.*, "Thread to strand binding of parallel network applications in massive multi-threaded systems," in *PPoPP 2010*.
[32] J. M. Richard McDougall, *Solaris internals: Solaris 10 and OpenSolaris kernel architecture*. Sun Microsystems Press/Prentice Hall, 2006.
[33] A. Settle *et al.*, "Architectural support for enhanced SMT job scheduling," in *PACT 2004*.
[34] D. Shelepov *et al.*, "Hass: A scheduler for heterogeneous multicore systems," in *ACM SIGOPS Operating Systems Review*, 2009.
[35] T. Sherwood *et al.*, "A Pipelined Memory Architecture for High Throughput Network Processors," in *ISCA 2003*.
[36] E. Shmueli *et al.*, "Evaluating the effect of replacing CNK with Linux on the compute-nodes of Blue Gene/L," in *ICS 2008*.
[37] A. Snavely *et al.*, "Symbiotic jobscheduling with priorities for a simultaneous multithreading processor," in *SIGMETRICS 2002*.
[38] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreaded processor," in *ASPLOS 2000*.
[39] D. Tam *et al.*, "Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors," in *EuroSys 2007*.
[40] D. M. Tullsen and J. A. Brown, "Handling long-latency loads in a simultaneous multithreading processor," in *MICRO 2001*.
[41] V. Čakarević *et al.*, "Characterizing the resource-sharing levels in the UltraSPARC T2 processor," in *MICRO 2009*.
[42] J. Verdú, *Analysis and Architectural Support for Parallel Stateful Packet Processing, PhD Thesis*. Universitat Politècnica de Catalunya, 2008.
[43] T. Wolf *et al.*, "Design considerations for network processor operating systems," in *ANCS 2005*.

Petar Radojković is a doctoral candidate in the Computer Architecture Department at the Technical University of Catalonia (UPC) and a researcher at the Barcelona Supercomputing Center, Spain. Petar received the M.Sc. degree in Computer Science from the University of Belgrade, Serbia, in 2006 and M.Sc. degree in Computer Architecture, Networks and Systems from the Technical University of Catalonia in 2009. His research interests include compilation and scheduling for multicore multithreaded architectures.

Vladimir Čakarević is a programmer, liberal, avid reader and occasionally a photographer. For the readers of this biography, assumable interested in less jocular part of his career, he received dipl. Ing. degree in Electric Engineering from Universty of Belgrade in 2006 and M.Sc. degree in Computer Architecture, Networks and Systems from Technical University of Catalonia in 2008. That did not make him much smarter, so he proceeded to doctoral studies in Computer Architecture at the same Spanish university. Currently, as a PhD student, Vladimir is affiliated with Barcelona Supercomputing Center. Vladimir worked on industry project with Sun Microsystems to facilitate scheduling and parallelization of network applications for their multi threaded processors. He spent three months at Sun Microsystems' headquarters in Menlo Park during 2008 and also worked at IBM Research Lab Haifa during three months in 2010, both times as an intern. For most of the time he is interested in parallel programming, network applications and multi threaded processors.

Javier Verdú is a tenure-track lecturer in the Computer Architecture Department at the Technical University of Catalonia (UPC), Spain. He has coauthored over 20 scientific international publications. His research interests include parallel architectures, optimization of multithreaded applications. Verdú received his PhD degree from the UPC in 2008, as well as his B.Sc. and M.Sc. degrees in computer science in 2001 from the University of Las Palmas de Gran Canaria, Spain.

Alex Pajuelo is an associate professor in the Computer Architecture Department at the Technical University of Catalonia (Spain). He received his M.Sc. degree in computer science in 1999 and his Ph.D. degree from the Technical University of Catalonia (Spain) in 2005. His research interests include performance evaluation methodologies, dynamic binary optimization and complex computing-demanding 3D visualization applications. He has coauthored more than 20 international publications. He has served in the organization of several international conferences as an external reviewer.

Francisco J. Cazorla is a researcher in the Spanish National Research Council. He leads the group on Computer Architecture/Operating System interface (CAOS) at the Barcelona Supercomputing Center (www.bsc.es/caos). He has worked in industrial projects (Intel, IBM, Sun Microsystems) as well as in European FP6 (SARC) and FP7 Projects (MERASA, PROARTIS, parMERASA). He has led two industrial projects, one with IBM and one with Sun Microsystems (now Oracle). He currently leads the PROARTIS project. He has two submitted patents on the area of hard-real time systems. His research area focuses on multithreaded architectures for both high-performance and real-time systems on which he is co-advising ten PhD theses. He has co-authored over 50 papers in international refereed conferences. He spent five months as a student intern in IBM's T.J. Watson in New York in 2004. He is member of HIPEAC and the ARTIST Networks of Excellence. Francisco J. Cazorla has been selected as one of the 100 Spanish 'leaders of the future' according to the May 2009 issue of the Capital Magazine. This issue seeks for the 100 young Spanish citizens that will most influence Spain's future in all innovation areas (www.capital.es). He has also been awarded by the Massachusetts Institute of Technology (MIT), as one of the 10 Spanish young innovators under 35 years, whose technical work has been successfully applied in recent years or has a great potential for development in the coming decades.

Mario Nemirovsky is an ICREA Research Professor at the Barcelona Supercomputing Center. He pioneered the concepts of Massively Multithreading (MMT) processing for the high performance processor and the by now well established Simultaneous Multithreding architecture (SMT). Mario was an adjunct professor at the University of California at Santa Barbara from 1991 to 1998. He has done research in many areas of computer architecture, including simultaneous multithreading, high-performance architectures, and real-time and network processors. Presently, he is working on Multicore/Multithreaded architectures, Network Processors and Architectures, Big Data issues, HPC and Cloud computing. Mario founded several USA companies ConSentry Networks, FlowStorm Networks; and XstreamLogic and since he has been in Barcelona he founded Miraveo and ViLynx. Before that, he was a chief architect at National Semiconductor, Apple Computers, Weitek and Delco Electronics, General Motors (GM). He architected GM engine control, being used in all GM cars for over 20 years. Mario holds 62 issued patents.

Mateo Valero has been a full professor in the Computer Architecture Department, Technical University of Catalonia (UPC), since 1983. Since May 2004, he is the director of the Barcelona Supercomputing Center (the National Center of Supercomputing in Spain). His research topics are centered in the area of high performance computer architectures. He has published approximately 500 papers, has served in the organization of more than 200 international conferences. His research has been recognized with several awards. Among them, the Eckert-Mauchly Award, Harry Goode Award, the "King Jaime I" in research and two National Awards on Informatics and on Engineering. He has been named Honorary Doctorate by the Universities of Chalmers, Belgrade, Las Palmas de Gran Canaria and Zaragoza in Spain and the University of Veracruz in Mexico. He is a fellow of the IEEE and the ACM and Intel Distinguished Research Fellow. He is an academic of the Royal Spanish Academy of Engineering, a correspondent academic of the Royal Spanish Academy of Sciences, an academic of the Royal Academy of Science and Arts and member of the Academia Europea.