# Thread to Strand Binding of Parallel Network Applications in Massive Multi-Threaded Systems

Petar Radojković[†] Vladimir Čakarević[†] Javier Verdú[‡] Alex Pajuelo[‡]
Francisco J. Cazorla[◊] Mario Nemirovsky[†⋆] Mateo Valero[†‡]

[†]Barcelona Supercomputing Center (BSC) [‡]Universitat Politècnica de Catalunya (UPC)
[⋆]ICREA Research Professor
[◊]Scientific Researcher in the Spanish National Research Council (CSIC)

{petar.radojkovic, vladimir.cakarevic, francisco.cazorla, mario.nemirovsky}@bsc.es
{jverdu, mpajuelo, mateo}@ac.upc.edu

## Abstract

In processors with several levels of hardware resource sharing, like CMPs in which each core is an SMT, the scheduling process becomes more complex than in processors with a single level of resource sharing, such as pure-SMT or pure-CMP processors. Once the operating system selects the set of applications to simultaneously schedule on the processor (workload), each application/thread must be assigned to one of the hardware contexts (strands). We call this last scheduling step the Thread to Strand Binding or TSB. In this paper, we show that the TSB impact on the performance of processors with several levels of shared resources is high. We measure a variation of up to 59% between different TSBs of real multithreaded network applications running on the UltraSPARC T2 processor which has three levels of resource sharing. In our view, this problem is going to be more acute in future multithreaded architectures comprising more cores, more contexts per core, and more levels of resource sharing.

We propose a resource-sharing aware TSB algorithm (TSBSched) that significantly facilitates the problem of thread to strand binding for software-pipelined applications, representative of multithreaded network applications. Our systematic approach encapsulates both, the characteristics of multithreaded processors under the study and the structure of the software pipelined applications. Once calibrated for a given processor architecture, our proposal does not require hardware knowledge on the side of the programmer, nor extensive profiling of the application. We validate our algorithm on the UltraSPARC T2 processor running a set of real multithreaded network applications on which we report improvements of up to 46% compared to the current state-of-the-art dynamic schedulers.

***Categories and Subject Descriptors*** C.4 [*Performance of Systems*]: Modeling techniques

***General Terms*** Algorithms, Measurement, Performance

***Keywords*** Process Scheduling, Simultaneous Multithreading, CMT, UltraSPARC T2

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*PPoPP'10,* January 9–14, 2010, Bangalore, India.
Copyright © 2010 ACM 978-1-60558-708-0/10/01. . . $10.00

## 1. Introduction

Current Multithreaded processors [1] exploit different Thread Level Parallelism (TLP) paradigms, such as SMT, FGMT or CMP. This allows better exploiting hardware resources and improves the overall system performance. For example, SMT or FGMT processors reduce fragmentation in the on-chip resources. However, increasing the number of hardware contexts (or strands) in SMT/FGMT processors is complex at the hardware level, which limits the number of strands that can be put in each CPU. This has motivated processor vendors to combine different TLP paradigms in their latest processors. For example, the Sun UltraSPARC T1 [1, 2] and T2 [3, 4], the IBM POWER5 [29] and POWER6 [22] and the Intel core i7 [7] combine two forms of TLP: CMP and FGMT in the case of Sun; and CMP and SMT in the case of IBM and Intel.

Combining several paradigms of TLP in a single chip introduces complexities at the software level, mainly in the job scheduling. In processors with a single resource-sharing level, such as pure-SMT and pure-CMP processor, the scheduling process is comprised of a single step. For example, assume that in a given computing system there are M runnable jobs. The Operating System (OS) job scheduler selects a set of N out of M jobs to compose the workload, where N is less or equal to the number of strands in the SMT (or the number of cores in the CMP). This procedure is known as co-schedule selection or workload composition [17]. However, in CMP processors comprised of SMT/FGMT cores, an additional step is required. Once the OS selects the applications/threads to schedule together in the processor (workload composition) each application/thread must be assigned to a particular strand of a given core [9]. We call this additional scheduling step the Thread to Strand Binding (TSB).

By choosing a given TSB we implicitly decide which hardware resources share the different running threads. In the IBM POWER5/6, Intel i7 and Sun UltraSPARC T1 there are two levels of resource sharing: resources shared among all strands of the processor (e.g., L2 or L3 cache, memory bandwidth, I/O); and resources shared among contexts of a given core (e.g., L1 cache, execution units, the register file). On the one hand, in the case of single-threaded applications, resource sharing should be kept at the lowest possible level and tasks that have to be placed together in a core should be the ones that conflict less – meaning, they use as

[1] In this paper we will use the term 'multithreaded processor' to refer to any processor executing more than one thread simultaneously. That is, Chip-Multiprocessors (CMP), Simultaneous Multithreading (SMT), Coarse-grain Multithreading (CGMT), Fine-Grain Multithreading (FGMT) or any combination of them are multithreaded processors.

little as possible the same hardware resources. On the other hand, in parallel applications, the benefit of mapping tasks that share data and/or code in the same core may largely overcome the negative effects of hardware resource interaction.

In current OSs, the TSB does not have a significant role when co-scheduling threads. Basically, the decision whether a job has to be scheduled in a given core depends on the cache and TLB affinity algorithms [8][31], which try to keep threads in the same virtual CPU in order to reduce cache and TLB misses as much as possible. Although these characteristics improve performance, they are not enough to fully exploit the capabilities of the current multithreaded processors with several levels of hardware resource sharing.

The importance of the TSB increases in existing and future multicore SMT/FGMT processors, mainly for parallel applications. In order to show this phenomenon we run on the UltraSPARC T2 a set of real multithreaded network applications. Our results show that the performance difference between the best and the worst TSB is up to 56% when running two instances of a networking application comprising a total of six threads; and up to 59% when running three instances of the same application (nine threads in total).

In this paper, we focus on software-pipelined network applications, in particular Layer2/Layer3 networking applications. These applications, such as IP Lookup, are critical services of routers, in either the edge or in the core of the Internet, comprising a main stage in the packet processing path. Most of these applications are executed in real time network-oriented systems that use low-overhead OSs to optimize the packet processing and to reduce the performance impact of OS noise due to the execution of management tasks [5, 6]. Currently, the TSB in such networking environments is done either by an standard resource-oblivious scheduling algorithm or by hand. The shared-resource oblivious scheduling provides less than optimal use of said resources, because it does not take into account the interaction among threads at the hardware level nor the communication between threads. Mapping threads by hand requires expertise on both, the processor architecture and the application itself. Moreover, to provide a good TSB, any change in the application or in the processor would require repeating the whole manual analysis and re-mapping.

In this paper, we present a systematic methodology to the TSB problem of software-pipelined parallel applications running in processors with more than one level of resource sharing. The main objectives of our methodology are: (1) Reducing the time it takes to find a good TSB that provides high overall performance; (2) Making it as much architecture independent as possible, so the same methodology can be targeted to different processor architectures; (3) Eliminating the need to understand which hardware resources the applications under study use.

By profiling a set of pre-determined TSBs in the target architecture we encapsulate both application and processor characteristics. The set is designed to measure the effects of conflicts in shared resources and benefits of inter-thread communication through the higher levels of cache. Once this initial profiling stage is done, in a second stage we use the gathered information to predict the performance of all possible TSBs of the application. The duration of the profiling and the prediction does not increase with the number of cores or application instances. In our experiments, the profiling and the prediction, needed for three instances of the network application (nine threads total) running on the UltraSPARC T2 processor, take less than 15 minutes. This is around 8000 times shorter than the time needed to run all possible TSBs. Moreover, our algorithm is able to accurately predict the actual best TSBs.

Based on the described methodology, we develop an off-line scheduler, *TSBSched*, for the Sun UltraSPARC T2 architecture. We validate our proposal with a real multithreaded Layer2/Layer3 network application: IP Forwarding (IPFwd), which is the most representative low layer network application and a critical process for any network system. Our results show up to 46% of performance improvement with respect to a scheduling algorithm that is not aware of the workload characteristics, and up to a 59% improvement with respect to a naive scheduling. Our TSBSched is able to select the absolute best TSB in 8 out of 14 workloads. In 5 of the remaining 6 benchmarks, TSBSched selects TSB with less than 3% of performance difference with respect to the absolute best TSB. In the worst case, our predicted best TSB introduces a performance degradation of 5% with respect to the best possible TSB.

The rest of this paper is structured as follows: Section 2 introduces basic knowledge of low layer network applications and the UltraSPARC T2 architecture. Section 3 describes the design and details of the TSBSched. In Section 4, we describe our experimental environment. We analyze the results of experiments in Section 5. Section 6 discusses related work. Finally, we conclude in Section 7.

## 2. Background

### 2.1 Layer2/Layer3 networking applications

Low-layer network applications (Layer2/Layer3) have become mandatory services of routers in the critical path of packet processing [23]. A good example is the lookup kernel, especially the IPv4/IPv6 packet forwarding, which is the most widely used protocol for Layer 3 communication. Actually, the packet forwarding workload is similar to most other low-layer network applications, the majority of them presenting the following execution pattern [23]: read some critical packet header values; search information in a global data structure; and trigger an action. Moreover, most network applications, especially the low-layer network applications [33], are partitioned into a number of stages (a.k.a. Software Pipelining) to optimize the locality of distributed caches and to increase the resource utilization of multicore SMT/FGMT processors.

Layer2/Layer3 networking applications exploit thread level parallelism following the two main approaches. In the first approach, software pipelining leads to multiple threads running different application stages that communicate each other. The first thread reads packets form the network device. Then the packet is sent to one or several processing stages. Once the packet is processed it is sent to the network device again. In the second approach, several instances of a particular application are executed simultaneously, as long as the packets can be processed independently, to provide enough computation power in the router to provide service under saturated network bandwidth.

Most low-layer network applications are stateless. That is, although the shared data structures are accessed during the packet processing, such data is not modified[2]. Thus, there is no data dependency among threads from different application instances during the packet processing (high thread level parallelism). At the same time, the data flow exists between pipeline stages in a given application instance, which may introduce performance degradation under a non-optimal workload partition [14].

Network applications have specific hardware and Operating System requirements.

**Hardware requirements:** Network processing requires high throughput to provide services to the increasing traffic level of the Internet. Massively multithreaded processors arise as the best alternative to exploit the multiple levels of parallelism exposed by network applications, where multiple threads can process several packets simultaneously.

[2] Global data structures, such as Lookup Table, are updated with very low frequency and has little impact on the system performance.

**Operating System requirements:** Real time network-oriented systems use low-overhead OSs to optimize the packet processing and to reduce the performance impact of OS noise due to execution of management tasks [5, 6]. These runtime environments provide static TSB that is applied at the beginning of the application image loading according to the configuration at compile time. Doing static TSB manually requires a deep knowledge of the requirements of each thread (several threads comprise a single pipelined network application) and the interaction among them at each level of shared resources in the architecture. Therefore, it is difficult to do an optimal TSB without previously analyzing an exponential number of experiments to collect all this information. Sub-optimal scheduling can lead to the router not providing enough throughput and therefore dropping packets.

### 2.2 The UltraSPARC T2 processor

The UltraSPARC T2 processor [3][4] is comprised of eight cores connected through a crossbar to a shared L2 cache (see Figure 1). Each of the cores has support for eight hardware contexts (strands) for a total of 64 tasks being executed simultaneously. Strands inside the 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*). In UltraSPARC T2 processor the overall performance of the system is more important than the performance of a single task, what results in a replication of a simple pipeline with a moderated performance.

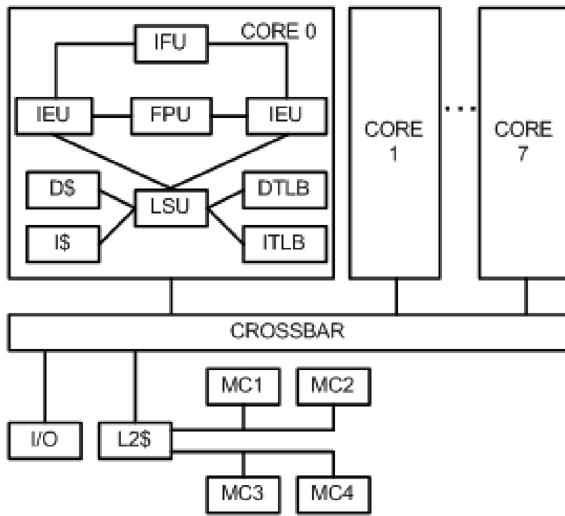



**Figure 1.** Schematic view of the three resource sharing levels of the Sun UltraSPARC T2 (CMT) processor

Processes simultaneously running on the Sun UltraSPARC T2 processor share (and compete for) different resources depending on how they are scheduled among strands. As it is shown in Figure 1, the resources of the processor are shared on three different levels: *IntraPipe*, among threads running in the same hardware pipeline; *IntraCore*, among threads running on the same core; and *InterCore*, among threads executing on different cores.

**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 among all processes that run on the same hardware core, the instruction fetch policy prevents any interaction between threads in different hardware pipes in the IFU. That is, the IFU actually behaves as two private IFUs, one for each hardware pipe.

**IntraCore**: Threads that run on the same core share the IntraCore resources: the 16 KB L1 instruction cache, the 8 KB L1 data cache (2 cycles access time), the Load Store Unit (LSU), the Floating Point and Graphic Unit (FPU), and the Cryptographic Processing Unit. Netra DPS low-overhead environment does not provide a virtual memory abstraction, the Instruction and Data TLBs are barely used, therefore we will exclude them from our analysis.

The FPU executes floating-point instructions, and integer multiplication and divide instructions.

**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 off-chip resources (such as I/O). The 4MB 16-way associative L2 cache has 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 DIMM (FBD) channels.

In order to fully utilize the performance of a massive multithreaded processors like the Sun UltraSPARC T2, it is important to understand which hardware resources are shared in each resource-sharing level. In the UltraSPARC T2, two threads running in the same hardware pipe conflict in all resource-sharing levels: IntraPipe, IntraCore, and InterCore. Threads running in two different hardware pipes of the same core conflict only at the IntraCore and InterCore levels. Finally, threads in different cores only interact 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 which has significantly higher access time.

## 3. TSBSched

TSBSched is a systematic methodology to approach the problem of optimal TSB for software-pipelined parallel applications running in processors with more than one level of resource sharing. In addition to reducing the time needed to find a good TSB, the two main objectives in the TSBSched design are the following:

**Making the scheduler as architecture independent as possible:** The only architectural data TSBSched requires is the hierarchy of different levels of resource sharing and the number of hardware contexts in each of them. For example, for the UltraSPARC T2 processor, TSBSched has to be aware of: (1) The three levels of resource sharing (IntraPipe, IntraCore, and InterCore); (2) That the processor contains eight cores, each of them has two hardware pipelines, and that each hardware pipeline has support for four hardware contexts. TSBSched does not require any information about which hardware resources are shared on which sharing level, nor the microarchitecture details of the processor resources (e.g., cache size, number and characteristics of execution units). The main benefits of making TSBSched microarchitecture independent are that it can be targeted to different processor architectures and that it does not require software developers to understand the processor's microarchitecture in detail.

**No need for a detailed knowledge of application's hardware requirements:** To select a good TSB, a scheduler has to be aware of the interaction among threads simultaneously running in the processor. In order to model these interactions, we run threads that are co-scheduled in different TSBs and measure the execution time of each thread [3]. The thread execution time encapsulates the

[3] For network applications, we measure the time needed to process a packet.

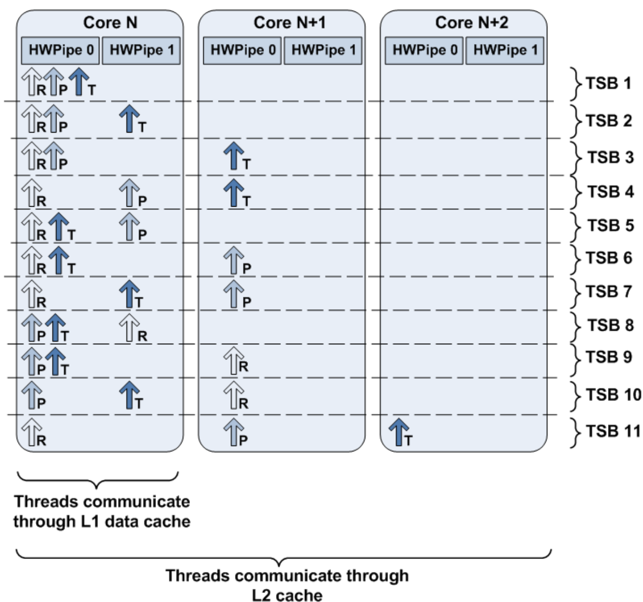



**Figure 2.** TSBs to measure the impact of data sharing on application performance – Input experiments for the *Base Time Table*

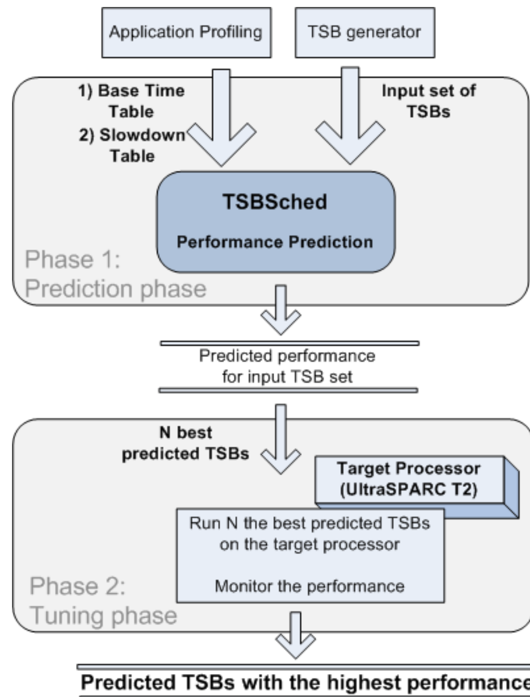



**Figure 3.** Schematic view of the major steps in TSBSched

information about the interference in hardware resources and the benefits of data and instructions sharing.

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 later predict the performance when several threads run simultaneously on a processor. In our methodology, we directly measure the interaction among concurrently running threads for a small set of TSBs (input experiments) and later use this data to model the interference in hardware resources among threads in any given TSB. The main benefit of our approach is that the scheduler does not need to be aware of application's characteristics. This information is encapsulated in the data passed to the scheduler. The information is gathered simply by measuring application execution time under different TSBs on the real hardware.

In TSBSched, we model two aspects of scheduling threads on processors with multiple levels of resource sharing: (1) The benefit of cache sharing when threads share data and instructions; (2) The collision in shared hardware resources.

### 3.1 Benefits of data and instructions sharing

Threads running on the same core in the UltraSPARC T2 processor share the L1 instruction and data cache. The L2 cache of the processor is shared among all concurrently running threads. If more threads share data or instructions, they may benefit from co-scheduling in the same processor core: one thread can prefetch data or instruction code to others reducing the number of L1 cache misses they experience.

In order to prove the accuracy of our model for applications that share instruction and data, we use a set of multithreaded software-pipelined applications. Each instance of the applications we use in our experiments consists of three stages (Receiver - R, Processing - P, and Transmitter - T) that comprise a *software pipeline*. Consecutive stages in software pipeline communicate through a shared data structure called *memory queue*. When consecutive software pipeline stages run in the same processor core, they communicate through L1 data cache [4]. When threads that share a memory queue execute in different cores, the communication is through L2 cache which causes additional L1 cache misses and the overhead in the application execution time.

The set of experiments we run in order to measure the impact of communication through L1 or L2 cache on application's performance is presented on Figure 2. We execute all possible TSBs (11 in total) of one application instance (comprised of R, P, and T threads) and measure the execution time of each thread. The execution time of a thread encapsulates the information about benefits of data and instructions sharing, but also about collision in shared hardware resources among threads that belong to the same application instance (software pipeline). The data gathered in these experiments are sorted in the *Base Time Table* that contains all data needed to fully describe the execution of one IPFwd instance in all possible TSBs. The *Base Time Table* is one of the inputs to TSBSched (see Figure 3).

### 3.2 Collision in shared hardware resources

Interferences at the hardware level between concurrently running threads depend on their characteristics, mostly which hardware resources they use. There is a trade-off between the model precision, the amount of input data and the time needed to gather it. In [32], the authors do a detailed characterization of the resource sharing levels in the UltraSPARC T2 processor. Some of the conclusions of this study apply, not only to UltraSPARC T2, but also to processors with several levels of resource sharing in general. Some of them are:

- The execution time of a process that dominantly stresses IntraPipe or IntraCore processor resources that is executed on a given core, is independent of the co-schedule of other processes as long as they execute on a different core (a remote core).

[4] We assume that the memory queue fits in the L1 data cache.

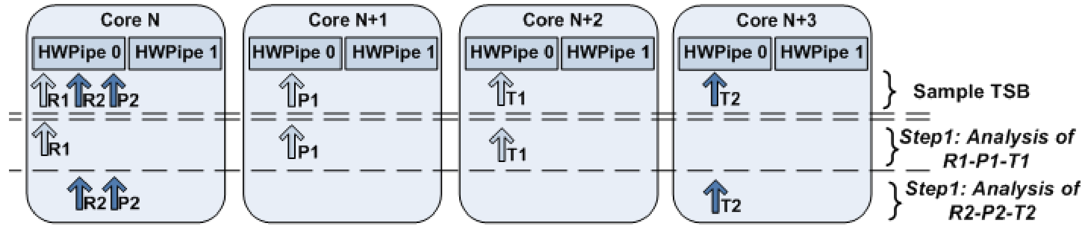



**Figure 4.** TSBSched prediction algorithm: Step 1

- The execution time of a process that stresses InterCore processor resources that is executed on a given core, is independent of the co-schedule of other processes as long as they execute on a remote core.

These facts dramatically reduce the complexity of the job scheduler: instead of analyzing all processes running on the processor, the job scheduler can reduce the scope of the analysis only to processes executing on the same core. For example, for a UltraSPARC T2 and a workload composed of more than 40 tasks, the number of TSBs that have to be analyzed reduces by dozens of orders of magnitude.

Based on this conclusion, in our model, we do not take into account the co-schedule of threads running on remote cores. In order to model the collision in hardware resources, we run each thread in the workload (*target thread*) with different combinations of other threads (*stressing threads*) on the same processor core and measure the execution time of the target threads. The data is sorted in the *Slowdown Table*, a data structure we use as one of the inputs to TSBSched (see Figure 3). For IPFwd application we use in our analysis, the *Slowdown Table* contains three groups of entries, one for each application thread – R, P, and T. In the experiments we run to characterize the R thread, we execute TSBs where one *target thread* R ($R_{tg}$) runs on the same processor core with all possible combinations of *stressing threads* $R_{st}$, $P_{st}$, and $T_{st}$. 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 $R_{tg}$ in the given TSB. The experiments we run to characterize $P_{tg}$ and $T_{tg}$ threads are analogous to $R_{tg}$ experiments.

In the current version of the TSBSched, for the presented set of applications, the profiling phase only takes several minutes to be run. Since this step is done off-line and only once for a given processor and a given set of applications, it is acceptable that the profiling part last much longer, hours or even a few days.

### 3.3 TSBSched functioning

TSBSched comprises two major phases: *Prediction phase* and *Tuning phase* (see Figure 3). First, in the Prediction phase, based on the input data, TSBSched predicts the performance for the set of input TSBs. The objective of TSBSched is not to predict the performance of each TSB, but to determine which are the TSBs with the highest performance. Later, in the Tuning phase, based on the predicted performance, the $N$ best (predicted) TSBs are executed on the real processor and the TSB with the highest measured performance is then selected as the final output of TSBSched.

**1. Prediction phase:** Based on the input data (the *Base Time Table* and the *Slowdown Table*), TSBSched predicts performance for the set of input TSBs. TSBs we use to validate TSBSched are composed of few IPFwd instances and each instance comprises R, P, and T threads. These TSBs are generated with a simple program we developed. The output of this phase is the list of input TSBs with the predicted performance for each of them.

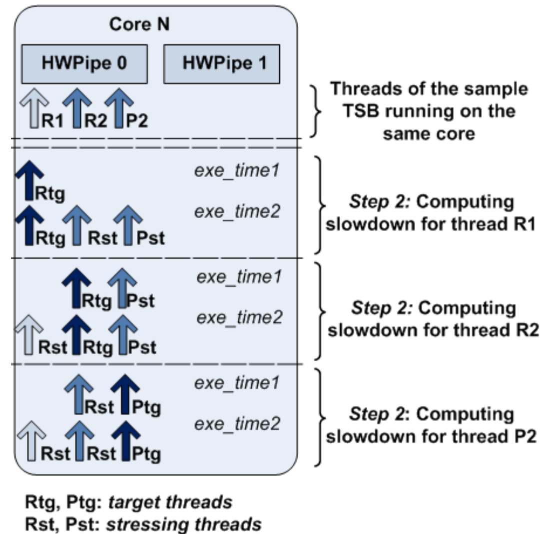



**Figure 5.** TSBSched prediction algorithm: Step 2

The TSBSched prediction phase has three steps:

1. In *Step 1*, we analyze each IPFwd instance in a TSB independently. We do not take into account the interference introduced by other applications running on the processor – we model only the interaction among threads that belong to the same application instance. In *Step 1*, we model the benefits of data and instruction sharing, and the collision in hardware resources among threads that belong to the same IPFwd instance.
2. In *Step 2*, we model the collision in hardware resources among all threads concurrently running on the processor. In this step, we take into account, not only threads that belong to the same application instance, but all threads in the TSB.
3. Finally, in *Step 3*, based on the analysis in *Step 1* and *Step 2*, we compute the predicted performance of the TSB.

Figure 4 presents an example TSB we use to explain our prediction algorithm. The TSB comprises two application instances: R1-P1-T1 and R2-P2-T2. Threads R1, R2, and P2 execute on the same hardware pipeline, while other threads execute on different processor cores. In *Step 1* of our prediction algorithm, we analyze application instances R1-P1-T1 and R2-P2-T2 independently. As input data, we use the *Base Time Table* since it contains execution times of R, P, and T threads of a single IPFwd instance running in all possible TSBs (see Section 3.1). First, we analyze R1-P1-T1.

For this example, we directly read the $basetime$ of R1, P1, and T1 from the fields of the *Base Time Table* that correspond to the *TSB11* presented on Figure 2. The *basetime* of R2, P2, and T2 threads is computed analogously.

In *Step 2*, we model the collision in hardware resources among concurrently running threads. As we explained in Section 3.2, we focus on interference among threads running on the same processor core. Since threads P1, T1, and T2 in the sample TSB execute alone on different processor cores, they do not suffer any slowdown because of collision in processor core resources. Only threads R1, R2, and P2 run on the same core, so in the *Step 2*, we focus on these threads. We compute: (i) The slowdown that R1 suffers because of R2 and P2; (ii) The slowdown that R2 suffers because of R1 and P2; and (iii) The slowdown that P2 suffers because of R1 and R2.

(i) Thread R1 executes on the processor core simultaneously with threads R2 and P2 (see Figure 5). In *Step 2*, we model the impact of threads R2 and P2 on thread R1: First, from the *Slowdown Table*, we read the execution time of *target* thread $R_{tg}$ when it runs alone on the same hardware pipe (*exe_time1*). Later, we read the execution time of *target* thread $R_{tg}$ when it runs with one $R_{st}$ and one $P_{st}$ threads on the same hardware pipe (*exe_time2*) – this correlates to the execution of R1 with R2 and P2 threads on the same hardware pipe. The difference between *exe_time2* and *exe_time1* is the slowdown R1 experiences because of interference with R2 and P2 in the sample TSB.

(ii) There is a small difference in algorithm if there are few threads from the same IPFwd instance running on the processor core. For example, thread R2 executes on the same hardware pipe with threads P2 and R1 (see Figure 5). Since R2 and P2 belong to the same application instance, in *Step 1* we completely model the interference between them. So, in *Step 2*, we model **only** the impact of thread R1 on thread R2: First, from the *Slowdown Table*, we read the execution time of *target* thread $R_{tg}$ when it runs with one *stressing* thread $P_{st}$ on the same hardware pipe (*exe_time1*) – this models the execution of R2 with thread P2 on the same hardware pipe. Later, we read the execution time of *target* thread $R_{tg}$ when it runs with one $R_{st}$ and one $P_{st}$ threads on the same hardware pipe (*exe_time2*) – this correlates to the execution of R2 with P2 and R1 threads on the same hardware pipe. The difference between *exe_time2* and *exe_time1* is the slowdown R2 experiences because of interference with R1 in the sample TSB.

(iii) The slowdown thread P2 experiences because of interference with R1 and R2 is computed analogously to *Step 2(ii)*.

In *Step 3*, we compute the predicted throughput of the TSB. First, we compute the execution time of every thread in the TSB. The thread's execution time is the sum of its *basetime* computed in *Step 1* and the *slowdown* from *Step 2*. Second, we compute the execution time of each IPFwd instance in a TSB. Since R, P, and T threads of IPFwd application process a packet in a pipeline, the execution time of the IPFwd instance is determined by the thread that has the longest execution time. At the end, from the IPFwd execution time, we derive the throughput of each IPFwd instance and the throughput of the TSB.

The number of input TSBs may be very large (tens or hundreds of thousands different TSBs per experiment). We pay special attention to make *Prediction phase* as fast as possible and to make it scale well with the number of TSBs. In the experiments presented in this paper, TSBSched predicts performance for up to 5000 TSBs per second [5]. This number is constant for different number of input TSBs, and it scales linearly with the number of threads in a TSB.

[5] TSBSched was running on Intel Dual Core processor running at the frequency of 1.83GHz and 1GB of memory.

**2. Tuning phase:** The objective of *Prediction phase* is not to predict the performance of each TSB, but to determine which are the TSBs with the highest performance. Later, in this phase, based on the predicted performance, the $N$ best (predicted) TSBs are executed on the real processor and the TSB with the highest measured performance is then selected as the final output of TSBSched. In the experiments we use to validate TSBSched, most of the times, running only ten best predicted TSBs is enough to capture the TSB that has the best actual (measured) performance. Running ten TSBs takes a couple of seconds on a target processor, for the processor and the application set we use in a study.

## 4. Experimental Environment

We validate TSBSched using a set of real multithreaded network applications running on an UltraSPARC T2 processor. Since the behavior of these applications is sensitive to network traffic characteristics, we pay special attention to describe Network Traffic Generator, the tool we use to generate representative network traffic. In order to avoid interferences between user applications and operating system processes, we run our experiments in Netra DPS a low-overhead environment [5, 6], which is also used to optimize network processing systems. We briefly describe Netra DPS being focused on the difference between this environment and a full-fledged operating systems, such as Linux or Solaris. At the end, we present the set of parallel network applications we use in experiments.

### 4.1 Hardware Environment

Our environment comprises two machines that manage the generation and processing of network traffic. One T5220 machine, comprising UltraSPARC T2 processor, runs the Network Traffic Generator (NTGen) [6] developed by Sun Microsystems. The NTGen sends the traffic through a 10Gb link to the second T5220 machine in which we run applications we use to validate TSBSched.

***Network Traffic Generator*** (NTGen) is a software tool, developed by Sun Microsystems, that is part of Netra DPS distribution [6]. It allows the synthetic generation of IPv4 TCP/UDP packets with configurable options to modify various packet header fields. In our environment, the tool modifies the source or/and destination IP addresses of 64Byte packets (representative for the worst case) following an incremental IP address distribution. NTGen generates enough traffic to saturate the network processing machine for three simultaneously running instances of the evaluated application (IP Forwarding).

### 4.2 Netra DPS

Real operating systems provide features, like the process scheduler or the virtual memory, to enhance the execution of user applications. But these features can introduce some overhead when measuring the performance of the underlying hardware since maintenance tasks of the operating systems are continuously interrupting the execution of user applications. For these reasons, in order to validate TSBSched, we use Netra DPS low overhead environment [5, 6]. Netra DPS provides less functionality than full-fledged OSs, but also introduces less overhead. In [25], the authors compare the overhead introduced by Linux, Solaris, and Netra DPS in the execution of benchmarks running on a Sun UltraSPARC T1 processor, showing 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 run-time process scheduler and performs no context switching. The mapping of processes to strands is performed statically at the compile time. It

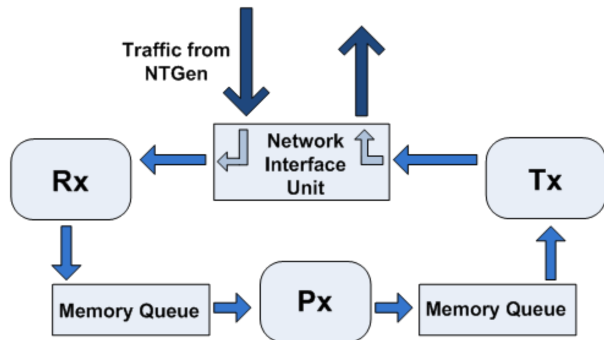



**Figure 6.** One instance of the IPFwd application

is the responsibility of the programmer to define the strand in which a particular task of the application will be executed. Netra DPS does not provide any interrupt handler nor daemons. A given task runs to completion on the assigned strand without any interruption.

### 4.3 Applications

We validate TSBSched for a set of real multithreaded network applications. Our experiments focus on the stateless Ethernet and IP network processing. One of the most representative of such applications is IP Forwarding (IPFwd), which is included in the Netra DPS package [6]. The IPFwd makes the decision to forward a packet to the next hop based on the destination IP address. The IPFwd instance we use in our experiments consists of three stages, each of them related to a different thread that form a software pipeline (see Figure 6):

- The receiver thread, or Rx, reads a packet from the Network Interface Unit (NIU) associated with the receiver 10Gb network link and writes the pointer to the packet into the memory queue that connects Rx and Px threads.
- The processing thread, or Px, 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 memory queue that connects the Px and Tx threads.
- Finally, the transmitter thread, or Tx, reads the pointer and sends the packet out to the network through the NIU associated to the sender 10Gb network link.

As IPFwd is a stateless application, the packets are processed without using critical sections in the Px stage (they do not modify any shared data structure). The IPFwd exploits packet level parallelism using two orthogonal approaches: (1) Replicating IPFwd instances that simultaneously execute on the processor and (2) Splitting one IPFwd instance in multiple stages that form a software pipeline.

The network features affect the performance of network applications. On the one hand, IPFwd is sensitive to the IP address distribution [19], since it affects the spatial locality of accesses to the lookup table (i.e. the table used to select the next hop of a given packet). On the other hand, IPFwd is also sensitive to the lookup table configuration. The routing table contents has an impact on the packet processing locality, while the table size determines the memory requirements (e.g. large routing tables present hundreds of thousands entries [16]).

We use three different IPFwd configurations to analyze complementary scenarios covering from the best to the worst case studies: (1) We make the lookup table fit in L1 data cache (*hash1-DL1*); (2) The table does not fit in L1 data cache, but it fits in the L2 cache (*hash1-L2*). The routing table entries are configured to cause a lot of data L1 misses in IPFwd traffic processing; (3) The table does not fit in L2 cache and the lookup table entries are initialized to make IPFwd continuously access the main memory (*hash1-Mem*). Thus, *hash1-DL1* is representative of the best-case, since it shows high locality in data cache accesses. However, *hash1-Mem* determines the worst-case assumptions used in network processing studies [28], in which there is no locality between packets doing the IP lookup.

We also validate TSBSched for a multithreaded low-layer network application that performs more complex packet processing. We design this application based on the IP Forwarding included in the Netra DPS package. In order to mimic more complex packet processing and multiple accesses to memory, 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 covering from the best to the worst case studies: (1) *hashN-DL1*: The lookup table fits in L1 data cache; (2) *hashN-L2*: The table does not fit in L1 data cache, but it fits in the L2 cache; (3) *hashN-Mem*: The table does not fit in L2 cache.

Finally, as a generic test aimed at representing a CPU-intensive network application (e.g. high layer packet decoding, URL decoding), we insert an assembly code containing 200 integer *add* instructions in a loop, which is repeated four times in Px stage of IPFwd application. This IPFwd variant, named *intADD*, fits the lookup table in L1 data cache, but it executes 800 additional integer *add* instructions making the Px a CPU stress processing stage. We use it to test how TSBSched handles partitioned network applications that have a high rate of resource conflicts on the IntraPipe shared level, because of stressing the instruction fetch and the integer execution units.

## 5. Evaluation

In this section, we evaluate TSBSched for different variants of IPFwd application. We run few IPFwd instances at once in different TSBs and measure the performance of each TSB. We compare actual, measured results to the ones we predict using TSBSched and show the performance difference between the actual best TSB and the best predicted ones. We also show the improvement our methodology obtains over a naive scheduling and a resource-oblivious state-of-the-art scheduling algorithm used in current OSs.

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

We present results for two and three IPFwd instances. We do not run experiments with more application instances, because our experimental environment is not able to provide enough traffic to saturate more IPFwd instances running at once. In configurations where the incoming traffic does not saturate the IPFwd maximum capacity, the number of TSBs that yield close-to-the-best result increases significantly. This would make the TSB selection problem much easier.

### 5.1 Two application instances

For the configurations with two IPFwd instances, we generate all possible TSBs. We execute each of them on the real hardware and measure the PPS when processing 3.6 million packets (long enough to get stable results). Based on these results, we select the best and the worst actual TSBs, which we use as reference points to validate our scheduling algorithm. On the other hand, we use TSBSched to predict the performance for all TSBs. We sort TSBs from highest to lowest predicted PPS (*Prediction Phase*) and execute top 10

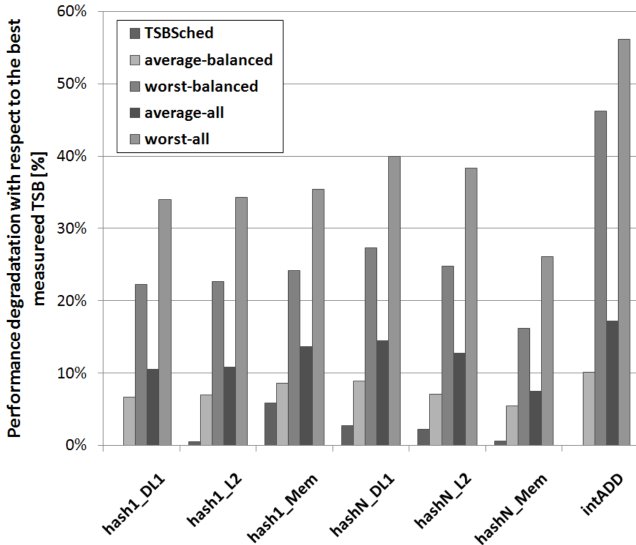



**Figure 7.** Performance degradation with respect to the best measured TSB (two IPFwd instances)

predicted TSBs on the real processor (*Tuning Phase*). The TSB with the highest measured PPS is then selected as the final output of TSBSched.

The results are presented on Figure 7. Different IPFwd variants are listed along X-axis of the figure while Y-axis shows performance degradation with respect to the best actual (measured) TSB. We present few groups of bars. *TSBSched* bar shows the performance difference between actual best TSB and the one predicted by TSBSched. Bars *average-all* and *worst-all* present the average and the worst performance degradation we measure for a given set of TSBs with respect to the actual best TSB. With these two values we aim to present the average and the maximum gain of TSBSched compared to the naive scheduling.

Bars *average-balanced* and *worst-balanced* present the average and the worst performance degradation with respect to the actual best TSB, but this time we take into account only balanced TSBs – TSBs that have threads equally distributed among processor cores and hardware pipes. Current state-of-the-art scheduling algorithms used in Linux and Solaris [10] [26] try to equally distribute running threads among different hardware domains (for UltraSPARC T2 processor, among processor cores and hardware pipes). The aim of this approach is to distribute running threads to equally stress hardware resources. Since current Linux and Solaris load balancing algorithms do not consider inter-thread dependencies nor different resource requirements of each thread, the performance of balanced TSBs are not the optimal. The values of *average-balanced* and *worst-balanced* bars present the average and the worst performance degradation incurred by a Linux-like scheduler.

The results presented on Figure 7 show that the accuracy of TSBSched is quite high in selecting the best possible TSBs. Out of seven configurations, we predict the absolute best in two configurations (*hash1_DL1* and *intADD*). In all cases but one (*hash1_Mem*) the degradation of the best predicted TSB with respect to the actual best TSB is less than 3%. The performance improvement with respect to state-of-the-art load balancers is between 5% and 10% in average, and it goes up to 46% in the worst case (*intADD*). The performance improvement with respect to naive process scheduler is between 10% and 17% in average; in the worst case it is between 26% and 56%.

**Table 1.** Performance degradation of the predicted best TSB for different number of processor cores (Two IPFwd instances)

| *best10* | 1 core | 2 cores | 3 cores | 4 cores | 5 cores |
|---|---|---|---|---|---|
| hash1_DL1 | 0.00% | 0.00% | 0.40% | 0.88% | 0.00% |
| hash1_L2 | 0.00% | 0.00% | 1.80% | 0.00% | 0.20% |
| hash1_Mem | 0.00% | 0.00% | 2.69% | 1.68% | 0.61% |
| hashN_DL1 | 0.00% | 0.00% | 2.83% | 2.06% | 0.42% |
| hashN_L2 | 0.00% | 0.00% | 0.94% | 1.71% | 0.00% |
| hashN_Mem | 0.00% | 0.59% | 0.69% | 0.29% | 0.92% |
| intADD | 0.00% | 0.00% | 0.00% | 0.00% | 0.00% |

In the previous example, we assume that all processor cores are available to run the IPFwd application. However, a more probable scheduling scenario is that the number of cores available to the workload is less than the number of processor cores because the virtualization is very common in network servers. In network systems, while certain logical domains (virtual machines) are used to run the data plane, which provides the network services to process the traffic (e.g. IP Forwarding), other domains are used to run the control plane, which provides management services (e.g. updating of Lookup table).

In Table 1, we present the analysis of the precision of TSBSched when only a subset of the cores in the UltraSPARC T2 can be used for running IPFwd. The left most column of the table lists the IPFwd application variant we use in the experiments. Results for different number of available processor cores are shown in different rows of the table. For each core count, we run the 10 best predicted TSBs and compute their performance degradation with respect to the actual best TSB which uses the set number of cores. We do not analyze TSBs using six cores as there is only one.

Out of 35 test cases, TSBSched is able to choose the best one in 19. Of the remaining 16, in 13 cases the performance is within 1% of the best TSB for selected number of cores. All results present the performance degradation of less than 3%. Furthermore, we have virtually perfect prediction for one and two processor cores, which should be the typical number of cores for two instances of IPFwd application. UltraSPARC T2 processor has eight strands per core and it is not likely that more than 20 strands will be dedicated to an application that has six threads.

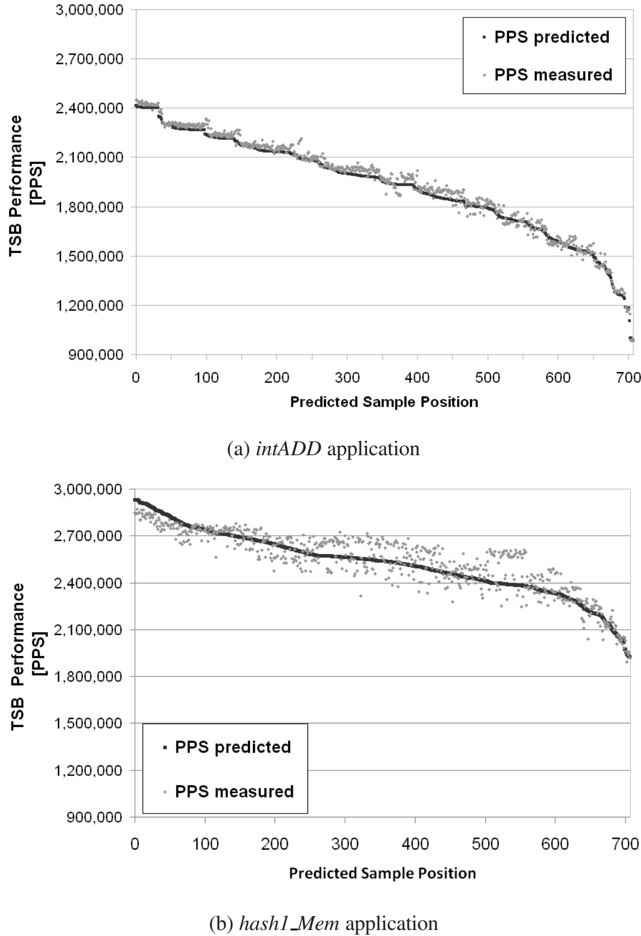
### 5.2 Three application instances

For two IPFwd instances, we run all possible TSBs on the processor and select the best and the worst actual TSBs, which we use as a reference points to validate TSBSched algorithm. As this configuration is comprised of six threads, there are 1526 different TSBs to execute. On our hardware [6] it takes around 10 seconds to execute a TSB, hence complete execution lasts a bit more than four hours. For three IPFwd instances, the number of different TSBs exceed half million, so it is unfeasible to run all of them on the processor because of execution time constrains.

#### 5.2.1 Statistical Sampling

As an alternative to the brute-force exploration we use systematic sampling [12]. The systematic sampling is a method of probabilistic sampling where the target population, all TSBs 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, then every $k$th element is selected until the end of the list. Where $k = \frac{\text{population size}}{\text{sample size}}$ and it is referred as the *sampling step*. Because

[6] We run the experiments on T5220 box that comprises an UltraSPARC T2 processor that operates on frequency of 1415MHz and 58GB of RAM




(a) *intADD* application

(b) *hash1_Mem* application

**Figure 8.** Comparison of predicted and measured performance for the chosen samples - (three IPFwd instances)

of execution time constraints, we choose a sample size of 700 elements which gives the sampling step $k = 750$.

We generate all possible TSBs and use TSBSched to predict their performance. Then we sort them by predicted performance and do the sampling. To make the technique effective, it is required to sort the population by a property that is correlated to the variable that is being measured. Ideally, we should sort the population by the value of the property that we want to measure. In our case, we sort the population by predicted PPS value. If we are able to show that the predicted value is closely related to the actual measured PPS value, we will prove the validity of our sample.

Figure 8 shows the PPS predicted with TSBSched algorithm and the actual PPS measured on the real hardware for samples chosen by the described technique. Figure 8(a) shows the results for *intADD* application which is selected because it has the biggest difference between the PPS provided by the best TSB and the PPS provided by the worst TSB. On the opposite end of the spectrum, in Figure 8(b) we show the results for *hash1_Mem* which has the lowest PPS difference between the best and worst TSB. The inserted addition operations in Px stage of *intADD* takes longer to execute than a L2 cache miss, and, as expected, the number of processed PPS is lower than in *hash1_Mem*.

As it may be observed, the prediction follows the actual measured results closely in both cases. The prediction precision is similar for the rest of IPFwd configurations. We conclude that the obtained samples are a valid representative of the actual applications performance. Note that we do not observe significant measured performance inversion with respect to the predicted results - the TSBs which performance is predicted to be in the top 5% are very unlikely to be found below 10% of the best measured results. We will use the obtained samples to assess the goodness of TSBSched in configurations with three IPFwd instances.

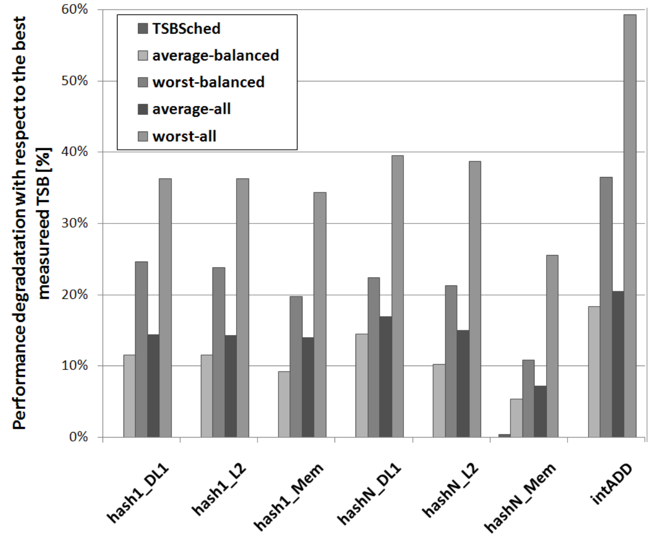



**Figure 9.** Performance degradation with respect to the best measured TSB (three IPFwd instances)

### 5.2.2 Results

The results for three IPFwd instances are shown in Figure 9. Different IPFwd variants are listed along X-axis of the figure while Y-axis shows performance degradation with respect to the best actual (measured) TSB. The group of bars are the same as on Figure 7: *TSBSched* bar shows the performance difference between actual best TSB and the one predicted by TSBSched; the values of *average-balanced* and *worst-balanced* bars present the average and the worst performance degradation incurred by Linux-like scheduler; the values of *average-all* and *worst-all* bars present the average and the worst performance degradation incurred by naive scheduling.

The results presented on Figure 9 show that the accuracy of TSBSched is even higher than for two IPFwd instances: out of seven configurations, TSBSched predicts the actual best TSB in six of them, all except *hashN_L2*. For *hashN_L2* application, the performance degradation of the best predicted TSB with respect to the actual best is only 0.4%. The performance improvement with respect to the state-of-the-art load balancers is between 5% and 18% in average and it goes up to 36% in the worst case (*intADD*). The performance improvement with respect to naive process scheduler is between 7% and 20% in average; in the worst case, it is between 25% and 59%.

We do not present per core analysis for three IPFwd instances as the used sampling technique do not include stratification per number of used cores, which would be necessary to exclude unwanted bias in the analysis.

## 6. Related Work

In architectures with a single-level of resource sharing, such as pure-SMT or pure-CMP processors, the job scheduler distributes

the workload by selecting N tasks from the M ready-to-run tasks in the system, where $M > N$ and $N$ is less than or equal to the number of contexts in the SMT (or the number of cores in the CMP). Once the workload is composed in a processor with a single-level of resource sharing, it does not matter how the threads are assigned to contexts in the SMT (or cores in the CMP). For example, assume a workload composed of threads {A,B,C,D} assigned to a given core is identified as the optimal schedule. The performance of this workload is independent of how the threads are scheduled inside a given core (e.g. {A,B,C,D}, {B,C,D,A}, {D,A,C,B}). Consequently, the works done so far in this area do not address the impact of thread scheduling on architectures with multiple-levels of shared resources.

There are numerous studies that explore the impact of thread workload selection on multicore or/and SMT/FGMT processors with a single-level of resource sharing. In [24] Parekh et al. propose to use microarchitectural level performance information as feedback to decide which threads to schedule in a SMT processor. The authors show the importance and the benefit of scheduling that takes into account process characteristics. In [30], the authors present the SOS scheduler that uses profile-based information to compose the workload. SOS runs several application mixes (workloads), examines the performance and applies heuristics to identify optimal schedules. The workloads that present the best symbiosis among combined tasks are selected.

Other approaches seen in pure SMT/pure CMP architectures [11, 15, 18], propose techniques to co-schedule threads that exhibit a good symbiosis in the shared cache levels and solve problems of cache contention. Kihm et al. [18] explore the design of a hardware cache monitoring system that provides enough information to co-schedule threads, at every scheduling interval on a simulated-based environment, to produce the least interference in the shared cache levels. Chandra et al. [11] present three performance models to predict the impact of cache sharing on bad co-scheduled threads, whose outputs can guide the OS scheduler decisions. In [15] the authors implement a cache-fair scheduling algorithm in the OS, which reduces co-runner-dependent performance variabilities by ensuring that the application always runs as quickly as it would under fair cache allocation, regardless of how the cache is actually allocated.

In [13], the authors classify applications regarding their usage of shared processor resources. The authors co-schedule applications with *base vectors* (i.e. micro-benchmarks that specifically stress a single CPU resource) and measure the slowdown application and base vector suffer. This data is used to predict the performance of any set of applications that simultaneously executes on the processor.

Other authors have proposed specific solutions for network applications to optimally map multithreaded workloads in parallel processors, specially in network processors. Kokku et al. [20] propose an algorithm to allocate network processing tasks to processor cores, mainly to reduce power consumption. In [33] the authors propose run-time support that considers the partitioning of applications across multiple cores. They address the problem of dynamic threads re-allocation due to network traffic variations, providing mapping solutions based on the application profiling and traffic analysis. However, they do not consider the different types of interaction among threads in the architecture.

All presented studies focus on defining a set of jobs/threads that will have the optimal performance when co-scheduled on the processor, taking into account a single level of shared resources. In our work, we show that, in order to determine optimal job/thread co-scheduling on multicore SMT processors with multiple-levels of resource sharing, in addition to workload selection, the one has to take into account the assignment of threads to strands, even inside a given core. We show that a given workload can have a significant performance difference depending on how threads are distributed among strands. We also propose TSBSched, an offline job scheduler that determines the optimal thread assignment for a set of processes simultaneously running on the processor.

Kumar et al. [21] and Shelepov et al. [27] take on the complex task of scheduling in heterogeneous cores architectures. These architectures are different than the UltraSPARC T2. Systems presented in [21] and [27] have multiple (mostly isolated) cores or even processors with different characteristic, while the UltraSPARC T2 has a set of cores and hardware pipes that have the same characteristics. Furthermore, the focus of these studies is finding an algorithm that matches application's hardware requirements with the processor core characteristics. However, the target cores are mostly independent, so there is no interferences between the threads running in the different cores. In our study, we explore interference among processes/threads depending on how they are distributed among homogeneous hardware domains (processor cores and hardware pipes) of a processor.

## 7. Conclusions

Integrating several TLP paradigms into the same die increases hardware resource utilization, and hence improves the overall system performance. This has motivated processors vendors to combine different TLP paradigms into their latest processors. Between others, the Sun UltraSPARC T1 and T2 combine CMP and FGMT, while the IBM POWER5, POWER6 and the Intel core i7 combine CMP and SMT.

However, this increase in processor resource utilization comes at the cost of introducing complexities at the software level, mainly in the job scheduling. We showed how in processors with several levels of resource sharing, in addition to the workload composition, an additional step (Thread to Strand Binding) is required. In this step each application/thread in the workload must be assigned to one of the execution cores. The TSB determines which hardware resources share the different running threads and has an increasingly high effect on performance when the number of cores and strands per core increases.

In this paper we have presented a systematic methodology to approach the problem of TSB for software-pipelined parallel applications running in processors with more than one level of resource sharing. Our methodology reduces the time it takes to find a good TSB that provides high system performance. At the same time, it removes the need for detailed knowledge of both the processor architecture and the hardware requirements of the applications being scheduled.

Our results show a reduction in the time it takes to find a good TSB of approximately four orders of magnitude for the workload comprised of three instances of a real network application consisting of three pipelined stages each. The reduction increases as the number of instances increases.

In experiments with real parallel network applications we show that our methodology is very accurate: we are able to find the best TSB in most cases, while in the all but one of the remaining cases TSBSched provides a TSB which performance is less than 3% worse than the best possible (3% is the determined precision of our measurements' environment). The single worst degradation of TSBSched with respect to the actual best TSB is within a 5% range.

## Acknowledgments

This work has been supported by the Ministry of Science and Technology of Spain under contracts TIN-2007-60625; Petar Radojković and Vladimir Čakarević have the FPU grant (Programa Nacional de Formación de Profesorado Universitario)of the Ministry

of Education of Spain. This work has been also supported by the HiPEAC European Network of Excellence and a Collaboration Agreement between Sun Microsystems and BSC. The authors wish to thank Jochen Behrens and Aron Silverton from Sun Microsystems for their technical support.

## References


[1] *OpenSPARC$^{TM}$ T1 Microarchitecture Specification*, 2006.

[2] *UltraSPARC T1$^{TM}$ Supplement to the UltraSPARC Architecture 2005*, 2006.

[3] *OpenSPARC$^{TM}$ T2 Core Microarchitecture Specification*, 2007.

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

[5] *Netra Data Plane Software Suite 2.0 Update 2 Reference Manual*, 2008.

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

[7] Intel 64 and IA-32 Architectures Software Developers Manual, 2009. `http://www.intel.com/Assets/PDF/manual/253665.pdf`.

[8] J. Aas. Understanding the Linux 2.6.8.1 CPU Scheduler. *SGI*, 2005. doi: http://josh.trancesoftware.com/linux/linux_cpu_scheduler.pdf.

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

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

[11] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA 05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 340–351. IEEE Computer Society, 2005.

[12] W. G. Cochran. *Sampling Techniques, 3rd edition*. Wiley-India, 2007. ISBN 8126515244.

[13] 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), in conjunction with ISCA-34*, 2007.

[14] R. Ennals, R. Sharp, and A. Mycroft. Task partitioning for multi-core network processors. In *In Compiler Construction*, pages 76–90, 2005.

[15] 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*, pages 25–38, 2007.

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

[17] R. Jain, C. Hughes, and S. Adve. Soft real-time scheduling on simultaneous multithreaded processors. *Proceedings of RTSS'2002*.

[18] J. Kihm, A. Settle, A. Janiszewski, and D. A. Connors. Understanding the impact of inter-thread cache interference on ILP in modern SMT processors. 7, 2005.

[19] 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 measurment*, pages 253–266, New York, NY, USA, 2002. ACM. ISBN 1-58113-603-X. doi: http://doi.acm.org/10.1145/637201.637242.

[20] 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):107–112, 2004. ISSN 0146-4833. doi: http://doi.acm.org/10.1145/972374.972393.

[21] R. Kumar, Dean M. Tullsen, Parathasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-ISA heterogenous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31st annual international symposium on Computer architecture*, page 64, 2004.

[22] 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 J. Res. Dev.*, 51(6), 2007.

[23] N. Shah. Understanding Network Processors. Technical report, EECS, University of California, Berkeley, Sept. 2001.

[24] S. Parekh, S. Eggers, H. Levy, and J. Lo. Thread-sensitive scheduling for SMT processors, 2000.

[25] P. Radojković, V. Cakarevic, J. Verdú, A. Pajuelo, R. Gioiosa, F. Cazorla, M. Nemirovsky, and M. Valero. Measuring Operating System Overhead on CMT Processors. In *SBAC-PAD '08: Proceedings of the 2008 20th International Symposium on Computer Architecture and High Performance Computing*. IEEE Computer Society, 2008. ISBN 978-0-7695-3423-7.

[26] J. M. Richard McDougall. *Solaris internals: Solaris 10 and OpenSolaris kernel architecture*. Sun Microsystems Press/Prentice Hall, 2006. ISBN 9780131482098.

[27] D. Shelepov, Juan Carlos Saez Alcaide, Stacey Jeffery, Alexandra Fedorova, Nestor Perez, Zhi Feng Huang, Sergey Blagodurov, and Viren Kumar. Hass: A scheduler for heterogeneous multicore systems. In *ACM SIGOPS Operating Systems Review*, pages 66–75, 2009.

[28] 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*, pages 288–299, New York, NY, USA, 2003. ACM. ISBN 0-7695-1945-8. doi: http://doi.acm.org/10.1145/859618.859652.

[29] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. *IBM J. Res. Dev.*, 49 (4/5), 2005.

[30] A. Snavely, Dean M. Tullsen, and Geoff Voelker. Symbiotic job-scheduling with priorities for a simultaneous multithreading processor. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 66–76, 2002.

[31] L. A. Torrey, J. Coleman, and B. P. Miller. A comparison of interactivity in the Linux 2.6 scheduler and an MLFQ scheduler. *Softw. Pract. Exper.*, 37(4):347–364, 2007. ISSN 0038-0644. doi: http://dx.doi.org/10.1002/spe.v37:4.

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

[33] T. Wolf, N. Weng, and C.-H. Tai. Design considerations for network processor operating systems. In *Proc. of ACM/IEEE Symposium on Architectures for Networking and Communication Systems (ANCS)*, Princeton, NJ, Oct. 2005.