

Characterizing the Resource-Sharing Levels in the UltraSPARC T2 Processor

Vladimir Čakarević¹, Petar Radojković¹, Javier Verdú², Alex Pajuelo²,
Francisco J. Cazorla¹, Mario Nemirovsky^{1,3}, Mateo Valero^{1,2}

¹ Barcelona Supercomputing Center (BSC) ² Universitat Politècnica de Catalunya (UPC)

³ICREA Research Professor

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

ABSTRACT

Thread level parallelism (TLP) has become a popular trend to improve processor performance, overcoming the limitations of extracting instruction level parallelism. Each TLP paradigm, such as Simultaneous Multithreading or Chip-Multiprocessors, provides different benefits, which has motivated processor vendors to combine several TLP paradigms in each chip design. Even if most of these combined-TLP designs are homogeneous, they present different levels of hardware resource sharing, which introduces complexities on the operating system scheduling and load balancing.

Commonly, processor designs provide two levels of resource sharing: **Inter-core** in which only the highest levels of the cache hierarchy are shared, and **Intra-core** in which most of the hardware resources of the core are shared. Recently, Sun Microsystems has released the UltraSPARC T2, a processor with three levels of hardware resource sharing: **InterCore**, **IntraCore**, and **IntraPipe**. In this work, we provide the first characterization of a three-level resource sharing processor, the UltraSPARC T2, and we show how multi-level resource sharing affects the operating system design. We further identify the most critical hardware resources in the T2 and the characteristics of applications that are not sensitive to resource sharing. Finally, we present a case study in which we run a real multithreaded network application, showing that a resource sharing aware scheduler can improve the system throughput up to 55%.

Categories and Subject Descriptors

C.1 [Processor Architectures]: Parallel Architectures;

C.4 [Performance of Systems]:

General Terms

Measurement, Performance

Keywords

Simultaneous Multithreading, Sun Niagara T2, CMT, Performance Characterization, CMP, Job Scheduling

1. INTRODUCTION

The limitations imposed when exploiting instruction-level parallelism (ILP) has motivated the appearance of thread-

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.

MICRO '09, December 12–16, 2009, New York, NY, USA.

Copyright 2009 ACM 978-1-60558-798-1/09/12 ...\$10.00.

Table 1: TLP paradigms in real chips. For each vendor, chips are shown in the chronological order

Chip	CMP	FGMT	SMT
IBM POWER4	✓		
IBM POWER5	✓		✓
IBM POWER6	✓		✓
Intel Pentium4			✓
Intel Core Duo	✓		
Intel Core i7	✓		✓
Sun UltraSPARC IV+	✓		
Sun UltraSPARC T1	✓	✓	
Sun UltraSPARC T2	✓	✓	

level parallelism (TLP) as a common strategy to improve processor performance. Multithreaded processors ¹ exploit different paradigms of TLP. On one extreme of the spectrum, threads executing in SMT processors [18][25] share most of the processor resources. On the other extreme, threads running on a CMP [15] share only some levels of the cache memory hierarchy. In between, there are other TLP paradigms like coarse-grain multithreading [4][23] or FGMT [11][21].

Each of these designs offer different benefits as they exploit TLP in different ways. For example, SMT or FGMT processors reduce fragmentation in on-chip resources. However, increasing the number of contexts in SMT/FGMT processors is complex at hardware level which limits the number of contexts in current SMT/FGMT CPUs. In order to make use of the available transistors, SMT/FGMT cores are replicated which allows overlapping long latency operations. This motivates processors' vendors to combine different TLP paradigms in their latest processors. Table 1 shows the state of the art processors and the forms of TLP they integrate. As may be seen, each vendor already incorporates several TLP paradigms in the same chip.

However, combining several paradigms of TLP in a single chip introduces complexities at the software level. Even if most current multithreaded processors are homogeneous, the combination of TLP paradigms leads to an *heterogeneous resource sharing* between running threads, i.e, the interaction between two threads varies depending on which resource sharing levels they have in common. Threads in the same core share many more resources, and hence interact much more than threads in different cores. This makes hardware resource sharing become a keypoint in the Operating System (OS) design.

¹In 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.

In current OSs, mechanisms like load balancing [3][24] are present to fairly distribute the load of the running processes among all the available logical CPUs (i.e., cores in a CMP architecture or contexts in a SMT architecture). Also, cache and TLB affinity algorithms [3][24] try to keep threads in the same virtual CPU in order to reduce cache misses and page faults 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. Therefore, in order to prevent continuous collisions in those shared resources that seriously degrade the overall system performance, the OS must have the knowledge of the resource sharing levels and the set of running applications. The results we present in the Section 5 show that a resource-aware job scheduler could improve the performance of real network applications running on a massive multithreaded processor up to 55% over a naive scheduler.

In this paper, we focus on the UltraSPARC T2 processor, which is a typical representative of the current and future trends in multicore and multithread designs. The T2 has eight cores, and each core has eight hardware contexts. In addition to InterCore and IntraCore hardware resource sharing, the T2 presents another *IntraPipe* level of resource sharing: in each core, the T2 groups the contexts into two hardware execution pipelines, from now on also referred as *hardware pipes* or *pipes*. Threads sharing the same hardware (execution) pipe share many more resources than threads in different hardware pipes.

In this paper, we make a complete characterization of the three resource-sharing levels of the T2. To that end, we create a set of microbenchmarks that stress particular hardware resources in a desired level. To reduce, as much as possible, any interference from external factors (like OS activities), we run the microbenchmarks on Netra DPS low-overhead environment [2]. Although Netra DPS provides less functionalities than full-fledged OSs, like Linux and Solaris, it introduces almost no overhead, which makes it ideal for our analysis. We also validate our findings with a real multithreaded network application that operates on individual packet headers to provide router forwarding functions (IP forwarding). This application is representative of the function level benchmarks that evaluate the performance of processors used for network processing [5]. From this characterization we extract important conclusions that affect the OS design, mainly the job scheduler and the load balancer. The main contributions of this paper are the following:

1. We make a detailed analysis of the resource-sharing levels of the UltraSPARC T2 processor. In order to quantify the benchmark interference in processor resources, we measure the *slowdown*. We define the *slowdown* as the relative difference between the execution time of a benchmark when it runs in isolation and when it shares processor resources with other benchmarks concurrently running on the processor. We identify which are the most critical shared hardware resources:

- ***IntraPipe***: The most critical shared resource at the IntraPipe level is the Instruction Fetch Unit (IFU). The Integer Execution Unit (IEU) is not a critical resource since most of the instructions executed in IEU have one-cycle latency. Our results show that the collision in the IFU can cause up to 4x slowdown in the applications that put high stress on this resource.

- ***IntraCore***: Usually, benchmarks comprised of pipelined instructions that execute in the FPU (integer multiplication, FP addition, FP multiplication, and others), have no interference in the FPU. On the other hand, benchmarks that execute non-pipelined FP or integer instructions suffer significant slowdown, up to 7.6x, when they are co-scheduled to share the FPU.

- ***InterCore***: At this level, we investigate the interference in two hardware resources: L2 cache and interconnection network. On one hand, we measure significant slowdown, up to 9.2x, when the applications have the collision in the L2 cache. On the other hand, the slowdown caused by collisions in the interconnection network and the limitations of the memory bandwidth is less than 15%.

2. On the application side, we show which characteristics of a program make it more or less sensitive to hardware resource sharing:

- ***IntraPipe***: Applications having low CPI are sensitive to resource sharing at IntraPipe level. This is because when low-CPI applications are co-scheduled to execute on the same hardware pipeline, they experience significant slowdown caused by the collisions in the IFU.

- ***IntraCore***: Applications that are comprised of non pipelined, long-latency operations that execute in the FPU (like integer or FP division) are sensitive to resource sharing at IntraCore level because of collision in the FPU.

- ***InterCore***: Collision in the L2 cache dramatically reduces application performance. Applications that use a large data set that fits in the L2 cache, are highly sensitive to the resource sharing at InterCore level.

3. For all microbenchmarks we define the set of the best, and the worst co-runners². We call the applications that significantly affect the other applications *high-interaction co-runners*, and the ones that hardly affect other applications *low-interaction co-runners*. Using this information, the OS job scheduler can distribute high-interaction processes on different pipes and cores of the processor in order to avoid the collision in hardware resources. On the other hand, sets of low-interaction processes can share IntraPipe and IntraCore processor resources having no or only negligible interference.

4. We show that, if a process dominantly stresses IntraPipe or IntraCore processor resources, its execution is independent of co-schedule of other processes as long as they execute on different cores. This dramatically reduces 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 more than 40 tasks running on eight T2 cores, the number of thread assignments that have to be analyzed reduces by dozens of orders of magnitude.

5. Also, we detect only negligible differences in execution time of InterCore tasks depending on the distribution of other tasks running on remote cores. This has two implications on an OS job scheduler design: (1) The slowdown due to interference in globally-shared resources may be significant and it depends on the characteristics and the number of running benchmarks. It is very important, when the job scheduler selects a set of tasks to concurrently run on

²Co-runners are the tasks that concurrently execute on the processor.

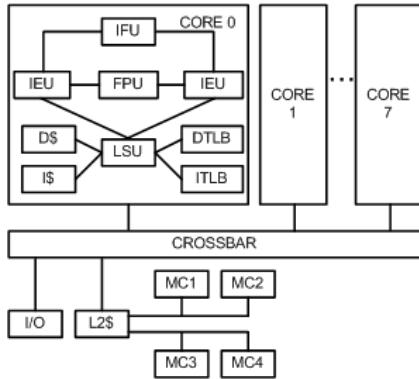


Figure 1: Schematic view of the Sun UltraSPARC T2 (CMT) processor

the processor, that it takes into account interference that selected tasks experience in the globally-shared resources. (2) On the other hand, once the workload is selected, the InterCore benchmarks interfere the same way regardless of how they are mapped to hardware contexts of the processor. The measured slowdown due to collision in globally-shared resources is the same regardless of whether or not tasks run in the same hardware pipe, different pipes, or in remote cores.

6. We make a case example with a real multithreaded network application in which we apply some of the concepts discussed in the previous characterization. We run several configurations of the application using different assignments of threads to hardware contexts and measure the system performance. We detect up to a 55% performance difference between the best and the worst thread assignments for a given workload. Our results also show that the performance difference increases with the number of concurrently running threads and the number of available processor cores. We predict that, with the trend of: more cores per processor, more hardware contexts per core, and more concurrently running processes that share the processor resources, the effect of the optimal process co-scheduling to system performance will increase, making the resource-aware process scheduler even more important.

The rest of this paper is structured as follows. Section 2 provides a detailed description of the UltraSPARC T2 processor resource sharing levels. In Section 3, we describe our experimental environment. Section 4 analyzes the behavior of the three different levels of resource sharing in the T2 processor. Section 5 presents insightful results for real network application. Section 6 relates all the previous work at literature. Finally, we conclude in Section 7.

2. THE ULTRASPARC T2 PROCESSOR

The UltraSPARC T2 processor [1] consists 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 amount of 64 tasks executing simultaneously. Strands inside a hardware core are divided into two groups of four strands, forming two hardware execution pipelines, from now on also referred as *hardware pipes* or *pipes*. The UltraSPARC T2 processor is representative of the current trend of processors in which the overall performance of the system is more important than the performance of a single task, resulting in a replication of cores with a moderated performance.

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 on 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 though 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. Thus, the IFU behaves as two private IFUs, one for each hardware pipe. When more threads running on the same pipe are able to fetch an instruction in the same cycle, the conflict is solved using a Least Recently Fetched (LRF) fetch policy. The IEU executes all integer arithmetic and logical operations except integer multiplications and divisions, which are executed in the Floating Point Unit (one per core).

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. Since Netra DPS low-overhead environment does not provide a virtual memory abstraction, the Instruction and Data TLBs are lightly used. We exclude them from our analysis.

The FPU executes integer multiplication and division operations, and all floating-point instructions. The FPU includes three execution pipelines: (1) FPX pipeline: FP add and multiplication instructions execute in the *Floating-point execution (FPX) pipeline*. These instructions are fully pipelined. They have a single cycle throughput and fixed execution latency (5 or 6 cycles) which is independent of the operand values. (2) FPD pipeline: Integer divide and square root instructions execute in the *Floating-point divide and square root (FPD) pipeline*. These instructions are not pipelined. FP divide and square instructions have a fixed latency of 19 and 33 cycles for single and double precision operands, respectively. Integer division latency depends on the operand values and it is between 12 and 41 cycles. (3) FGX pipeline: Graphic instructions execute in the *Graphics execution (FGX) pipeline*.

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. The L2 cache access time is 22 cycles, and the L2 miss that accesses the main memory lasts on average 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 the T2, two threads running in the same pipe conflict in all resource-sharing levels: IntraPipe, IntraCore and InterCore. Threads running in two different pipes of the same core conflict only at the IntraCore and InterCore levels. Finally, threads in different cores only interact at InterCore level.

Resource Sharing Level	Benchmark	Stress on Execution Units	Pipelined?	CPI in Isolation
IntraPipe	intadd	IEU	YES	1
	intmul	IEU + FPU	YES	5
IntraCore	intdiv	IEU + FPU	NO	20
	FPadd	IEU + FPU	YES	6
	FPdiv	IEU + FPU	NO	33
	Dcache	LSU + L1 Dcache	YES	2
InterCore	L2cache	LSU + L1 Dcache + Crossbar + L2 cache	YES	22
	mem	LSU + L1 Dcache + Crossbar + L2 cache + mem controllers	YES	185

Table 2: Microbenchmarks used in this paper

3. EXPERIMENTAL ENVIRONMENT

In this section, we explain the software and the hardware environment we use in order to characterize the UltraSPARC T2 processor. We pay special attention to provide measurements without interferences from external factors, like OS activities. In all experiments we measure either the real execution time or the Packets Per Second (PPS) of reference applications.

3.1 Netra DPS

Real operating systems provide features, like the process scheduler or the virtual memory, to enhance the execution of user applications. However these features can introduce some overheads 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, to characterize the UltraSPARC T2 we use the Netra DPS low overhead environment [2]. Netra DPS provides less functionalities than a full-fledged OSs, but also introduces less overhead. In [17], we 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 runtime process scheduler and performs no context switching. The mapping of processes to strands is performed statically at compile time. Also, Netra DPS does not provide any interrupt handler nor daemons.

3.2 Microbenchmarks

In a multithreaded processor, the performance of a program depends, not only on the processor architecture, but also on the other applications running at the same time on the same processor and their specific execution phases. Under such conditions, evaluating all the possible programs and all their phase combinations is simply not feasible. Moreover, since real applications present several complex phases that stress different resources, it would be impossible to attribute a fine-grain performance gain or loss to a particular interaction in a given shared resource level. For this reason, we use a set of synthetic benchmarks (*microbenchmarks*) to stress a particular hardware resource since this provides a uniform characterization based on the specific program characteristics and avoids collateral undesired results.

We define three groups of microbenchmarks: *IntraPipe*, *IntraCore*, and *InterCore*. The idea of each microbenchmark is to mostly execute instructions that stress a given hardware resource level. For example, *InterCore* microbenchmarks are

mostly comprised of instructions that execute in hardware resources shared at core level (e.g. FPU or L1 data cache). For each resource level we create several microbenchmarks. Because of space constraints, for each resource sharing level, we choose few representatives that show different behavior. The same conclusion we obtain for benchmarks we run applies to the other microbenchmarks in the same group.

Table 2(a) summarizes the set of designed microbenchmarks as well as the resource sharing level they focus on. From left to right, the first column of the table defines the resource sharing level the benchmark is designed to stress. The following two columns list the microbenchmarks in our study and the processor resources each of them stresses. The fourth column describes whether the benchmarks run instructions that are executed in a pipelined hardware resource. The last column shows the CPI of each microbenchmark when it executes in isolation in the T2 processor.

IntraPipe microbenchmarks stress resources that are private to each hardware pipe of the processor: IEU and IFU. In order to measure the effects of IntraPipe resource sharing, we design the *intadd* microbenchmark. It consists of a sequence of integer add (*intadd*) instructions (although *intsub*, *and*, *or*, *xor*, *shift*, and other integer arithmetic and logic instructions are also valid), having a CPI equal to 1.

IntraCore microbenchmarks can be divided into three subgroups: *integer*, *floating point (FP)*, and *data cache benchmarks*. The integer IntraCore benchmarks, *intmul* and *intdiv*, comprise a sequence of integer multiplication and integer division instructions, respectively. *FP IntraCore* benchmarks, *FPadd* and *FPDIV*, consist of FP addition and FP division instructions, respectively. Integer multiplication and division, and FP addition and division instructions execute in Floating Point Unit (FPU) that is shared among all processes running on the same core.

The *Dcache* benchmark traverses a 2KB data array that fits in L1 data cache with a constant stride of one cache line (16Bytes). The *Dcache* benchmark consists of a sequence of load instructions that access different cache lines inside the L1 data cache.

InterCore microbenchmarks stress resources shared among all the cores in the processor (crossbar, L2 cache, and memory subsystem). We design two *InterCore* microbenchmarks: *L2cache* and *mem*, both of them stressing the memory subsystem of the processor. The *L2cache* benchmark traverses a 1.5MB data array that fits in L2 cache with a constant stride of one L2 cache line (64Bytes). The *L2cache* benchmark consists of a sequence of *load* instructions that access different banks and different cache lines inside the L2 cache. The *mem* benchmark traverses a 6MB data array with a constant stride of one L2 cache line (64Bytes). Since the 6MB array does not fit in L2 cache, *mem* benchmark always misses in the L2 cache, accessing the off-chip memory.

All microbenchmarks are written directly in UltraSPARC ISA. All benchmarks are designed using the same principle: A sequence of instructions of the targeted type ended with the decrement of an integer register and a non-zero backward branch to the beginning of the loop. The assembly functions are inlined inside a C program that defines the number of *iterations* for the assembly loop. All the microbenchmarks are compiled and linked with Sun Studio 12 without optimizations to prevent the compiler from applying any code optimizations, changing the core of the loop. The overhead of the loop and the calling code is less than 1% (more than

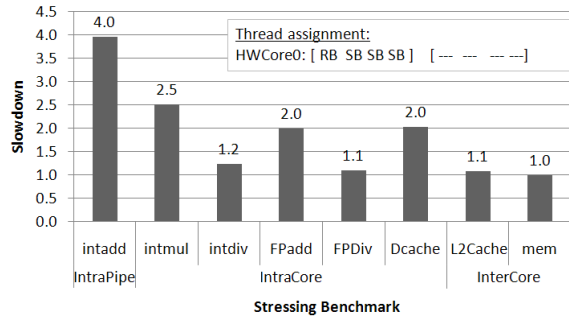


Figure 2: Slowdown of the *intadd* microbenchmark (reference benchmark) caused by three stressing benchmarks running on the same hardware pipe

99% of the time the processor executes only the desired instruction).

Benchmarks that access cache memory, *Dcache* and *L2cache*, are implemented using the concept of pointer chasing. An array is allocated in a contiguous section of memory. Then, each array entry is initialized with the address of the array element that will be accessed next. The distance in memory between these elements, that are to be accessed one after another, is exactly the size of the cache line. At runtime, the benchmark executes a sequence of indirect load instructions, accessing the following cache line in every instruction.

3.3 Methodology

In order to measure the benchmark execution time we read the *tick* counter register. After the benchmark design and implementation, we validate their behavior in order to guarantee that the benchmark fulfills the desired features. We use the Netra DPS profiling tools [2] to access the hardware performance counters in order to monitor the microarchitectural events (such as the number of executed instructions, number of FP instructions, number of L1 and L2 cache misses) during the benchmark execution.

To obtain reliable measurements in the characterization of the UltraSPARC T2 processor, we use the FAME (FAirly MEasuring Multithreaded Architectures) methodology [26]. This methodology ensures that every program in a multi-programmed workload is completely represented in the measurements. For this, the methodology advises to re-execute once and again one program in the workload until the average accumulated IPC of that program is similar to the IPC of that program when the workload reaches a steady state. FAME determines how many times each benchmark in a multi-threaded workload has to be executed so that the difference between the obtained average IPC and the steady state IPC is below a particular threshold. This threshold is called MAIV (Maximum Allowable IPC Variation). The execution of the entire workload stops when all benchmarks have executed as many times as needed to accomplish a given MAIV value. For the benchmarks used in this paper, in order to accomplish a MAIV of 1%, each benchmark must be repeated at least 5 times.

4. MICROBENCHMARK RESULTS

In this section, we show the whole characterization of the UltraSPARC T2 processor based on the levels of resource sharing. We present the results of IntraPipe, IntraCore, and InterCore benchmarks when they run on the same pipe,

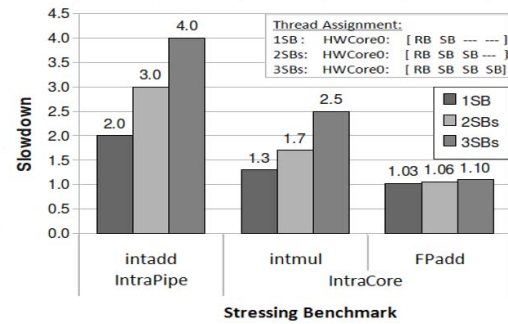


Figure 3: Slowdown of the *intadd* microbenchmark as we increase the number of stressing benchmarks in the same hardware pipe

same core, and different cores. In order to quantify the benchmark interference, we measure the *slowdown*, the relative difference between the benchmark execution time when it shares processor resources with other benchmarks concurrently running on the processor and when it runs in isolation

$$slowdown = \frac{execution\ time_{multitask}}{execution\ time_{isolation}}$$

4.1 IntraPipe benchmarks (*intadd*)

In order to explore the impact of IntraPipe resource sharing on the performance of *intadd*, we run *intadd* as the Reference Benchmark (RB) on the same pipe with three instances of an stressing benchmark (SB), which represents the largest number of SBs that can run in the same hardware pipe. The results are presented in Figure 2. The different stressing benchmarks are listed along the X-axis. The Y-axis shows the slowdown of the reference benchmark. We observe that the slowdown of the *intadd* benchmark is inversely proportional to the CPI of the stressing benchmark. The highest slowdown is measured when *intadd* executes with three more *intadd* stressing benchmarks having a CPI of 1. In this case, *intadd* reference benchmark gets the access to the IFU once every four cycles, what leads to the slowdown of 4x. The slowdown that *intadd* RB suffers is lower when it concurrently executes with the *intmul* and *FPadd* stressing benchmarks (2.5x and 2x respectively), and it is negligible for the *intdiv* and *FPdiv* stressing benchmarks comprised of long latency instructions. The same conclusion (the slowdown is inversely proportional to the CPI of the stressing benchmark) remains when running *intadd* versus microbenchmarks that stress the memory subsystem. We measure 2x slowdown when the *intadd* runs simultaneously with the *Dcache* benchmark that accesses the L1 data cache (few-cycle instruction latency), only negligible slowdown when executed together with the *L2cache* benchmark (having CPI of 22), and no slowdown at all with the *mem* stressing benchmark (having CPI of 185). We conclude that IntraPipe benchmarks are affected only by sharing the IFU with low-CPI InterCore and IntraPipe benchmarks. We measured no effect of the sharing of other hardware resources even if reference and stressing benchmarks execute on the same hardware pipe.

Figure 3 shows the slowdown of *intadd* running on the same pipe with one, two, and three instances of pipelined stressing benchmarks: *intadd*, *intmul*, and a non-pipelined benchmark *FPadd*. For all stressing benchmarks, the slowdown of the *intadd* reference benchmark increases approximately with the number of co-runners. In the case of the *in-*

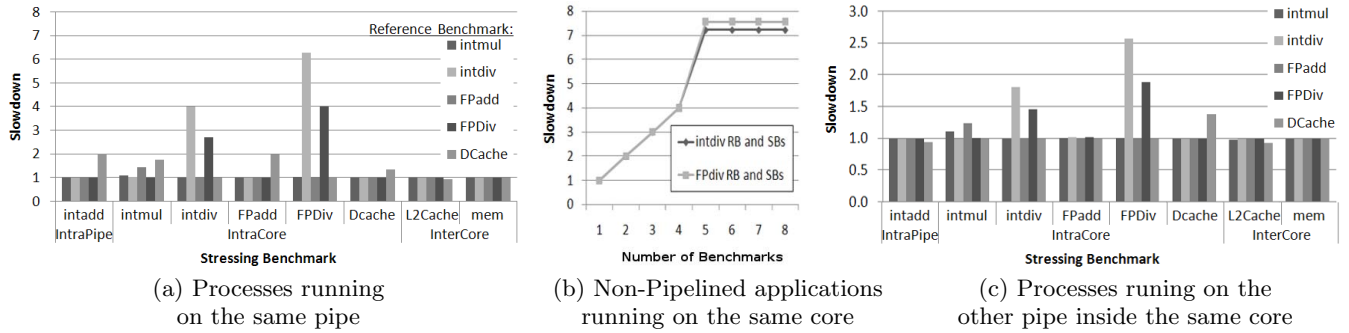


Figure 4: Impact of microbenchmarks on IntraCore benchmarks

intadd stressing benchmark, the IFU is equally shared among the reference and all stressing threads. Reference benchmark *intadd* suffers a slowdown exactly proportional to the number of stressing benchmarks. When *intadd* runs with *intmul*, the IFU is not equally shared among all processes using it. The IFU of the Sun UltraSPARC T2 processor uses a Least Recently Fetched (LRF) fetch policy. Since *intmul* benchmark has a CPI of 5 (it fetches one new instruction every five cycles) and *intadd* has a CPI equal to 1 (it can fetch one instruction every cycle), all collisions in IFU between them are resolved giving higher priority to *intmul*, while *intadd* has to wait for the next cycle. This means that when *intadd* simultaneously runs with a single instance of *intmul* benchmark, every five cycles (when *intmul* is ready to fetch an instruction), *intadd* is stalled for one cycle because of the collision in IFU. Out of five cycles, *intadd* will fetch one instruction (and later execute) in only four. This implies the slowdown of $5/4 = 1.25x$. When there are two *intmul* stressing benchmarks, *intadd* will be able to fetch instructions three out of five cycles what leads to the slowdown of $5/3 = 1.67x$. Following the same reasoning, when *intadd* shares the IFU with three *intmul* stressing benchmarks, *intadd* suffers the slowdown of $5/2 = 2.5x$ comparing to its single threaded execution.

The third set of bars in Figure 3 presents the slowdown *intadd* experiences when it runs with one, two, and three instances of *FPdiv* stressing benchmark. The slowdown of *intadd* is significantly lower. This is because *FPdiv* benchmark consists of long latency instructions (tens of cycles) that hardly stress the IFU. The slowdown that *FPdiv* benchmark introduce in the execution of *intadd* is nearly independent on the number of co-runners. The explanation for this is that the *FPdiv* instructions are not pipelined, so for most of the time the strand in which they execute is stalled waiting for the results of the computations.

The IntraPipe benchmarks do not experience any interaction or slowdown coming from other benchmarks executed in a different hardware pipe or core. The slowdown of IntraPipe benchmark is caused by the sharing of the IFU and the IEU. If there are no co-runners in the same pipe, all the available bandwidth of these two units is devoted to the execution of the IntraPipe benchmark.

4.2 IntraCore Benchmarks (*intmul*, *intdiv*, *FPadd*, *FPdiv*, *Dcache*)

In addition to IntraCore processor resources (i.e. FPU and L1 caches) IntraCore processes also stress the IFU and IEU as all instructions that are executed in the FPU have to pass through one of the two IEUs in each core. When IntraCore benchmarks execute on the same pipe with the stressing

benchmarks, they present some slowdown caused by: (1) the interference in the IFU and IEU (like IntraPipe benchmarks) and (2) the interference in the IntraCore resources. Next we present results that show both effects.

In order to explore the effect of IntraPipe resource sharing on IntraCore benchmarks, we run them on the same pipe with three stressing benchmarks. The results are presented on Figure 4(a).

IntraPipe resource sharing: The *Dcache* benchmark is an example of an IntraCore benchmark that can be affected by IFU sharing. When the *Dcache* executes on the same pipe with three instances of *intadd*, *intmul*, *FPadd*, and *Dcache* benchmarks, it experiences up to 2x slowdown because of the collision in the IFU. The explanation for this slowdown is the same as in the case of IntraPipe reference benchmarks: the IFU is able to fetch only one instruction per cycle and per hardware pipe. The probability of a collision in the IFU is lower with a higher CPI.

FPU sharing: The slowdown because of FPU sharing is significantly different for different benchmarks depending on the type of the instructions they execute. On one hand, *intmul* and *FPadd* benchmarks that consist of instructions that are pipelined in the FPU, experience a moderate slowdown (11% and 44% respectively) when they run with three *intmul* benchmarks on the same pipe. On the other hand, *FPdiv*, the benchmark containing non-pipelined FP instructions suffers a slowdown of 2.7x and 4x when it runs on the same pipe with *intdiv* and *FPdiv*, respectively. The slowdown is worse when *intdiv* runs with *intdiv* and *FPdiv*, 4x and 6.3x respectively.

In order to better explore the effect of FPU sharing among pipelined and non-pipelined instructions, we run up to eight instances of benchmarks that stress FPU on the same core and observe their interference.

Pipelined benchmarks, *intmul* and *FPadd*, both have 6 cycles of latency and a throughput of one instruction per cycle. As long as the number of benchmarks running on the same core is less or equal to six, we detect no slowdown. When there are seven and eight pipelined benchmarks running on the core, the FPU is not able to serve all concurrently running tasks, so there is a slowdown of 17% and 34% for seven and eight stressing benchmarks, respectively. We conclude that it is unlikely that pipelined instructions executing in the FPU (like integer multiplication, FP addition, or FP multiplication) suffer any slowdown because of the collision in the FPU.

Figure 4(b) shows the slowdown for the *intdiv* and *FPdiv* reference benchmarks that execute non-pipelined FP instructions. The X-axis of this figures presents different layouts of stressing benchmarks. A layout is defined by the

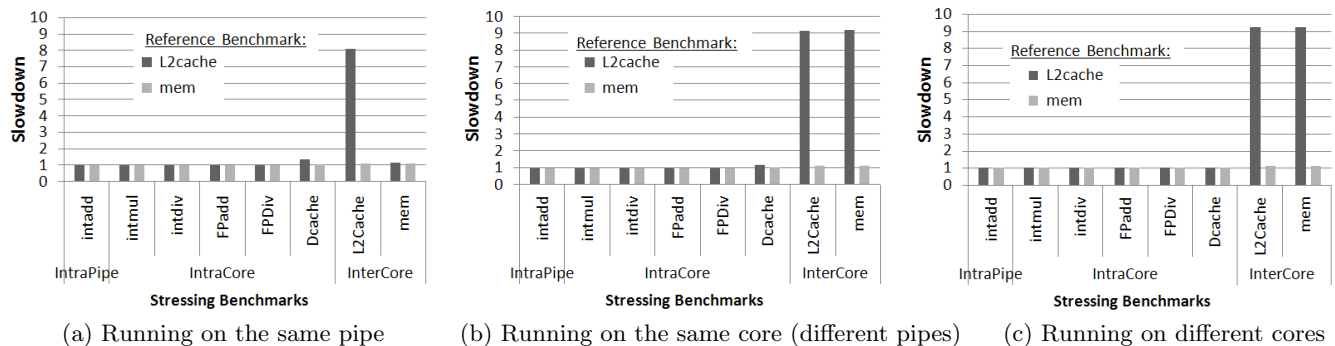


Figure 5: Impact of microbenchmarks on InterCore benchmarks

total number of microbenchmarks in it: In the layout with one benchmark the microbenchmark is executed alone in the first pipe (*pipe 0*) of the core, in the one with four benchmarks all of them are executed in *pipe 0*, in the five-task layout there are four tasks in *pipe 0* and one in *pipe 1*. The measurements are always taken from the benchmark running in *pipe 0*. FPU can serve only one non-pipelined instruction at a time, while all other instructions of the same type are stalled. For example, when there are several *FPdiv* benchmarks running on the same pipe, FPU is equally time-shared among all of them, causing a slowdown in the reference benchmark proportional to the total number of tasks. When there are two benchmarks running at the same time (one reference and one stressing), *FPdiv* will use FPU only 50% of time (slowdown of 2x). The same reasoning is applicable for the 3x and 4x slowdown when there are two and three stressing benchmarks running on the same pipe with the reference one. However, when we schedule additional stressing benchmarks to run on the other pipe inside the same core, *FPdiv* does not experience a linear slowdown with the number of running tasks. When there are stressing benchmarks running on the other pipe, the slowdown of *FPdiv* is 7.6x and it is independent on the number of additional stressing benchmarks. The explanations for these results is that tasks running on the same pipe, access the FPU through a FIFO queue, and because of that the slowdown is equally distributed among all tasks. When tasks run on different pipes inside the same core, they are divided in two groups based on the pipe a task is running in (tasks running on the same pipe form the group). The FPU is equally shared among groups no matter how many tasks there are in each group (as long as there is at least one). When we add one stressing benchmark on the other pipe, the slowdown of *FPdiv* increases from 4x to 7.6x. Since FPU is equally shared among groups, increasing the number of stressing benchmarks running on the pipe 1 will not cause additional slowdown to *FPdiv* running on the pipe 0. The same reasoning is valid for *intdiv* results presented in Figure 4(b).

IntraCore resource sharing: In order to explore the effect of IntraCore resource sharing on IntraCore benchmarks, we run them with four stressing benchmarks running on different hardware pipe inside the same core. The results are presented on Figure 4(c). Pipelined IntraCore benchmarks, *intmul* and *FPadd* experience a moderate slowdown, 11% and 25% respectively, when they are co-scheduled to run with *intmul* stressing benchmarks. FPU is able to concurrently process up to six pipelined instructions and it has a throughput of one instruction per cycle. Since pipelined in-

structions from different tasks can simultaneously execute in the FPU experiencing almost no interference, the slowdown we detect is modest. We measure higher slowdown on non-pipelined reference benchmarks. The *intdiv* benchmark presents slowdowns of 1.8x and 2.6x when it is stressed with *intdiv* and *FPdiv* benchmarks. When *FPdiv* is executed simultaneously with four *intdiv* and *FPdiv* stressing benchmarks running on the other pipe, it experiences a slowdown of 1.5x and 1.9x, respectively. When more non-pipelined instructions execute in the FPU, they are serialized, so the slowdown they suffer is significant. On the other hand, we detect lower slowdown with respect to the case when non-pipelined benchmarks run in the same hardware pipe. This is because, as we have already noted in the previous paragraph, the FPU is shared on two different levels: First, among groups of threads that run on different hardware pipes using Round Robin policy; later among threads running on the same pipe using FIFO policy. Hence, the interference among threads assigned to different pipes is lower than when they run on the same pipe.

The *Dcache* benchmark experiences a slowdown only when it is co-scheduled to run with another *Dcache* stressing benchmarks. When the reference *Dcache* is co-scheduled to run on the same core with four stressing *Dcache* benchmarks we measure a slowdown of 40%. Data sets of five *Dcache* benchmarks running on the same core (one reference and four stressing) do not fit in L1 data cache, what generates capacity L1 cache misses that cause the slowdown.

The IntraCore benchmarks do not experience any interaction or slowdown coming from other benchmarks executed in different processor cores. This comes from the fact that IntraCore benchmark slowdown is caused by sharing the FPU and the L1 caches and running without co-runners on the same core ensures that all the available bandwidth of these resources is devoted to the execution of this benchmark.

4.3 InterCore Benchmarks

InterCore benchmarks stress global processor resources shared among all tasks executing on the processor: L2 cache, on-chip interconnection network (crossbar), and interface to off-chip resources (such as memory or I/O).

The interference that applications have in the L2 cache depends on many factors: data footprint of the applications, memory access frequency, cache size, cache organization, cache line replacement policy, etc. [8]. Such an analysis is out of the scope of our study. We rather focus on understanding how L2 cache intensive applications affect other non-cache-intensive tasks (e.g. tasks stressing integer or FP execution units).

Figure 5 presents the results we obtain when we run the InterCore benchmarks, *L2cache* and *mem*, co-scheduled with: (a) three stressing benchmarks running on the same hardware pipe; (b) four stressing benchmarks running on the same hardware core, but on different pipe; (c) eight stressing benchmarks running on another core. The different stressing benchmarks are listed along the X-axis of the Figures 5(a), (b) and (c), and the Y-axis shows the slowdown of the reference benchmarks. We observe that, the only IntraPipe and IntraCore benchmarks that cause some slowdown to *L2cache* or *mem* are the ones that use the L1 data cache: the *L2cache* presents slowdowns of 38% and 19% when it runs with the *Dcache* benchmarks on the same hardware pipe and core, respectively. This slowdown is the consequence of sharing the Load Store Unit (LSU) among tasks running on the same core. On Figure 5(c), there is no interference among *L2cache* and *Dcache* benchmarks since they execute on different hardware cores and, hence, they use different LSUs.

There are two main reasons why InterCore benchmarks experience a slowdown when they simultaneously run with other InterCore applications: (1) Collision in the L2 cache, and (2) Collision in the interconnection network and the interface to off-chip resources (memory).

1. Collision in the L2 cache: When the *L2cache* benchmark runs versus three *mem* stressing benchmarks on the same pipe, it experiences only a slowdown of 10% (see Figure 5(a)). But, when there are four *mem* replicas running on the other pipe or eight *mem* replicas running on the other core, the slowdown raises up to 9.2x (see Figures 5(b) and (c)).

This slowdown is due to the fact that the working set of four or more copies of *L2cache* does not fit in L2 cache, what causes capacity L2 misses. We also observe that the slowdown reference benchmark suffers is the same in all three cases: when stressing benchmarks run on the same pipe, same core, or different cores

2. Collision in the interconnection network and interface to off-chip resources: We design the *mem* benchmark to miss in L2 cache on every memory reference even when it runs in isolation, so it cannot experience any slowdown because of additional L2 cache misses. The slowdown we measure for *mem* reference benchmark when it is co-scheduled to run with InterCore stressing benchmarks is due to collisions in the interconnection network and the memory bandwidth. We obtain a slight slowdown (between 10% and 15%) in all cases: *L2 cache* and *mem* stressing benchmarks, and IntraPipe, IntraCore, and InterCore resource sharing (see Figure 5). Again, the slowdown is almost the same in all three cases: when benchmarks run on the same pipe, same core, or different cores.

An interesting observation is that the slowdown we measure does not depend on how InterCore benchmarks are scheduled in the processor. In order to explore how task co-scheduling interferes the globally-shared processor resources, we run five copies of the *L2cache* in three different thread assignments and measure the slowdown for each *L2cache* instance. In *Thread Assignment 1*, we run one copy of the *L2cache*, marked *R*, alone on the *core 0*, while all other copies, marked *S*, are running inside the same pipe of the *core 1* - $\{[R][S][S][S][S]\}$. In *Thread Assignment 2*, we keep one copy of the *L2cache* alone on the *core 0*, and we distribute pairs of benchmarks among two pipes of the *core 1*

- $\{[R][S][S][S][S]\}$. In *Thread Assignment 3*, one copy of the *L2cache* is still running alone on the *core 0*, but this time, we distribute the remaining four benchmarks on different hardware pipes of *core 1* and *core 2* - $\{[R][S][S][S][S]\}$. The thread assignments we use cover all the relative co-schedules of the tasks on the processor: running on the same pipe, same core, and different cores. We also cover two extreme cases: (1) When tasks run alone on the core, they share processor resources only at the InterCore level; (2) Four benchmarks running on the same hardware pipe (*Thread Assignment 1*), in addition to sharing resources on the IntraCore and InterCore level, present the worst case of InterPipe resource sharing³.

The slowdown that *L2cache* benchmarks suffer, 9.2x in all assignments, is the consequence of the interference in the L2 cache, but also in the on-chip interconnection network and the memory bandwidth. More importantly, the slowdown we measure is the same for all benchmarks in all thread assignments – it is independent on the way the benchmarks are scheduled on the processor. We conclude that the interference in the last resource-sharing level (globally-shared resources) is independent on thread assignment.

4.4 Implications on Job Scheduling

State-of-the-art job schedulers provide several mechanisms, such as load balancing and cache affinity, to exploit as much as possible, the resource sharing capabilities of the underlying hardware. But, in order to make an optimal scheduling, the operating system job scheduler has to be aware of the interference among tasks co-scheduled to simultaneously execute on the processor. Since there may be many simultaneously running tasks, each of them having different phases that stress different processor resources, the problem of determining an optimal co-scheduling may be very complex even for single- or dual- core processors with only few hardware contexts on each core. Even worse, the complexity of the problem scales rapidly with the number of concurrently running tasks, the number of hardware contexts on a core, and number of cores on the processor.

The results presented in our study can be used to decrease the complexity when determining a good task co-scheduling:

1. Defining good co-runners: Base on the type of the resources a task use, we may predict how well it will run when mapped on the same pipe/core with other tasks.

IntraPipe: These applications may experience significant slowdown because of collision in the IFU with other tasks running on the same pipe. IntraPipe *high-interaction* co-runners are other low-CPI IntraPipe applications. Much better co-runners of IntraPipe benchmarks are IntraCore and InterCore tasks having higher CPI. IntraPipe applications do not experience any slowdown when they are co-scheduled to run on the same pipe with applications that consist of long latency instructions, such as non-pipelined integer and FP instructions, and instructions accessing L2 cache or the main memory. Applications comprised of long latency instructions are *low-interaction* co-runners to IntraPipe applications.

IntraCore: We measure no slowdown when IntraCore benchmarks execute on the same hardware pipe or core with InterCore benchmarks. *High-interaction* co-runners of the non-pipelined IntraCore benchmarks are non-pipelined

³The highest number of tasks that can concurrently run on a hardware pipe is four.

IntraCore benchmarks themselves. *High-interaction* co-runners of pipelined IntraCore benchmarks are IntraPipe and pipelined IntraCore benchmarks. InterCore applications are *low interaction* co-runners of IntraCore tasks.

InterCore: Running more instances of InterCore benchmarks can cause significant slowdown with respect to their execution in isolation. *Low-interaction* co-runners to InterCore applications are IntraPipe and IntraCore applications, since they share almost no hardware resources. Based on this, the OS job scheduler can distribute high-interaction tasks on different pipes and cores of the processor in order to prevent collisions in hardware resources. On the other hand, low-interaction tasks can share IntraPipe and IntraCore processor resources having no or negligible interference.

2. Reducing the scope of the analysis: Our results show that the execution of an IntraPipe or IntraCore tasks running on the *Core A* is independent of co-schedule of other tasks as long as they execute on a core different from *Core A* (a remote core). For InterCore tasks, we detect only negligible difference in execution time depending on the way other tasks are assigned to remote cores. This may lead to a very important conclusion which dramatically reduces the complexity of the job scheduler: Instead of analyzing all tasks running on the processor, the job scheduler can reduce the scope of the analysis to tasks executing on the same core, or even on the same pipe (for IntraPipe reference application). For example, let's assume there are nine independent tasks that should be scheduled on up to eight cores of the T2 processor. When the OS job scheduler takes into account relations among all tasks, no matter if they run on the same core or not, there are around 520 000 different thread assignments to analyze. But, if the job scheduler assumes only negligible interference depending on a way other tasks are assigned to remote cores, the number of thread assignments to analyze is reduced to 8,500, what is the reduction of more than 60x. The number of thread assignments that can be excluded from analysis using this assumption increases exponentially with the number of concurrently running tasks and the number of available processor cores: for example, for more than 40 tasks running on eight T2 cores, the scale of the problem would be reduced by dozens of orders of magnitude.

3. Sharing of global resources: Based on the results we present for InterCore benchmarks, we derive two important conclusions that have implications to OS process scheduler design: (1) The slowdown because of interference in globally-shared resources may be significant and it depends on the characteristics and the number of running benchmarks: we measure a slowdown of 9.2x when *L2cache*, as a reference benchmarks, does not hit in L2 cache. Hence, it is very important that when the job scheduler selects a set of tasks that concurrently run on the processor (the workload), it takes into account the interference the selected workload experiences in the globally-shared resources. (2) On the other hand, once the workload is selected, the InterCore benchmarks interfere the same way regardless of their schedule to hardware contexts of the processor: the slowdown we measure because of collisions in globally-shared resources is the same no matter if tasks run in the same hardware pipe, different pipes, or in remote cores. Our results confirm the conclusions of the [10] where authors observe the independence between the number of L2 misses and thread assignments with a set of commercial server benchmarks.

The conclusion we show in this section are based on the results we presented for homogeneous microbenchmarks that stress only specific processor resources. Even so, we argue that real applications having different phases and stressing different resources, can be modeled using the presented set of microbenchmarks as base. For example, the behavior of each application phase can be approximated and analyzed as the behavior of one or several microbenchmarks. Thus, the conclusions we present can be applied to schedule any set of applications on multicore multithreaded processor.

5. REAL NETWORK APPLICATIONS

In this section, we explain the network application we use in this paper, the hardware equipment required to run the application, and the main conclusions we obtained when an application runs on the UltraSPARC T2 processor.

5.1 Hardware Environment

Our environment comprises two machines that manage the generation and processing of network traffic. One T5220 machine (Sun UltraSPARC T2 processor) runs the Network Traffic Generator (NTGen) [2] developed by Sun. The NTGen sends the traffic through a 10Gb link to the second T5220 machine, in which we run the IP Forwarding (IPFwd) application.

Network Traffic Generator (NTGen) is a software tool, developed by Sun, that is part of Netra DPS distribution [2]. It allows the generation of synthetic IPv4 TCP/UDP packets with configurable options to modify various packet header fields. In our environment, the tool modifies the source IP addresses of 64Byte packets following an incremental IP address distribution. The processor running IPFwd receives over 18 million packets per second from NTGen through the 10Gb link that is enough to saturate the network processing machine even when there are several IPFwd instances running simultaneously.

5.2 Network Applications

Our network experiments focus on the 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. IPFwd makes the decision to forward a packet to the next hop based on the destination IP address.

The network features affect the performance of network applications. On the one hand, IPFwd is sensitive to the IP address distribution [13], since it affects the spatial locality of accesses on 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 may have hundreds of thousands entries).

5.3 Experimental Results

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 (IPFwd-DL1); (2) The table does not fit in L1 data cache, but it fits in the L2 cache (IPFwd-L2). The routing table entries are configured to cause a lot of DL1 misses in IPFwd traffic processing; (3) The table does not fit in L2 cache and the lookup table entries are initialized to make

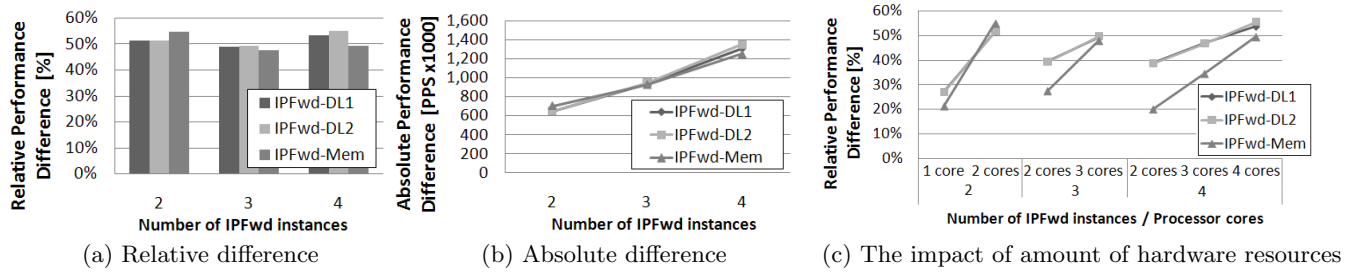


Figure 6: Performance difference between the best and the worst thread assignments of IPFwd

IPFwd continuously access the main memory (IPFwd-Mem). Thus, IPFwd-DL1 is representative of the best-case, since it shows high locality in data cache accesses. However, IPFwd-Mem determines the worst-case assumptions used in network processing studies [20], in which there is no locality among packets in the IP lookup.

An IPFwd instance consists of three processing stages, each of them mapped to a different thread:

- The receiver thread, or Rx, reads the network packet from the Network Interface Unit (NIU) associated to the receiver 10Gb network link and writes the pointer to a 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 another 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.

In this section, we present results that show how the performance obtained by a real network application (IPFwd) tightly depends on the thread distribution on the processor. We run two, three, and four instances of different IPFwd configurations (IPFwd-DL1, IPFwd-DL2, and IPFwd-Mem) and measure the system throughput (number of Packets processed Per Second (PPS)) for different thread assignments.

In experiments with two IPFwd instances (six threads in total), we run all possible thread assignments. However, the experiments with three and four instances of the application present too many possible assignments to run all of them (e.g. more than 500.000 different assignments for three IPFwd instances). In order to overcome this problem, according to the analysis of the application characteristics, we manually select the thread assignments that present good and bad performance. Since, we do not run all thread assignments but only a sample, the performance difference between the best and the worst co-schedule is at least as large as the one we present. In experiments with two IPFwd instances, we schedule the threads to one or two processor cores. We run three IPFwd instances on up to three processor cores, and four instances on up to four cores. Further increment in the number of cores that applications can use does not affect the system performance, but decreases the utilization of the processor, so we exclude these results from our analysis.

Figure 6(a) presents the performance difference between the best and the worst assignments for IPFwd-DL1, IPFwd-DL2, and IPFwd-Mem applications. The X-axis shows the number of application instances simultaneously running on the processor. The Y-axis presents the relative performance difference between the best and the worst thread assignment. As it is shown on the figure, we measure the significant per-

formance difference (between 48% and 55%) for all three configurations of IPFwd benchmark, in all three cases: two, three, and four application instances running on the processor.

In Figure 6(b), we present the data for the same set of experiments as for Figure 6(a), but this time we measure the absolute performance difference between the best and the worst thread assignment. We see that the absolute difference increases linearly with the number of tasks running on the processor: 650.000 Packets Per Second (PPS) for two instances of IPFwd; 950.000 and 1.300.000 PPS of difference for three and four instances, respectively.

Figure 6(c) shows the relation between the amount of available hardware resources (cores) and the performance difference between different thread assignments. In addition to the number of IPFwd replicas simultaneously running on the processor, the X-axis shows the number of processor cores used by the applications. The Y-axis presents the relative performance difference between the best and the worst assignment. For all IPFwd configurations (IPFwd-DL1, IPFwd-DL2, and IPFwd-Mem) the performance difference increases with the number of cores the application uses.

Even the results presented on Figure 6 clearly show a significant performance difference among different thread assignments, the one may argue that the performance difference is the consequence of non-balanced thread distribution over different hardware domains (cores, pipes) of the processor. In order to show that the equal distribution alone is not enough to provide the optimal thread assignment, we repeat the experiments presented on Figure 6, but this time taking into account only the assignments in which threads are evenly distributed among processor cores and hardware pipes (they are balanced). The results are presented on Figures 7(a) and (b). Figure 7(a) presents the performance difference between the best and the worst balanced assignments for IPFwd-DL1, IPFwd-DL2, and IPFwd-Mem applications. The X-axis shows the number of instances of IPFwd simultaneously running on the processor. The Y-axis presents the relative performance difference between the best and the worst balanced assignments. We see that, even if the equal thread distribution decreases the best-to-worst-assignment performance gap, the difference we measure is higher than 10% in most of the cases, and it goes up to 17% (4 IPFwd instances, IPFwd-DL1 and IPFwd-DL2 applications). In Figure 7(b), we present the absolute performance difference for the same set of experiments as for the Figure 7(a). This time, again, we take into account only thread assignments that are equally distributed over hardware domains (cores, pipes) of the processor. The results we present show the same as when we take into account all thread assignments (balanced and non-balanced): the absolute performance dif-

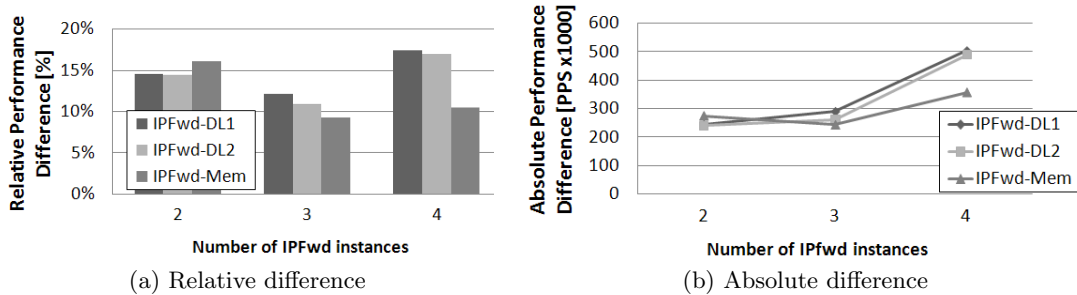


Figure 7: Performance difference between the best and the worst balanced thread assignments of IPFwd

ference increases with the number of tasks concurrently running on the processor.

Based on the results presented on Figures 6 and 7, we can derive the following conclusions:

First, we show that a resource-aware job scheduler can significantly improve the performance of network applications running on massive multithreaded processors. In our case, we measure up to 55% higher throughput when threads are assigned to hardware contexts taking into account different levels of resource sharing.

Second, we show that the absolute performance difference increases with the number of threads simultaneously running on the processor, see Figures 6(b) and 7(b), and with the amount of available hardware resources, see Figure 6(c). In the future, with more cores per processor, more hardware contexts per core, and more concurrently running tasks that share the processor resources, the effect of the optimal thread co-scheduling in system performance will increase what will make resource-aware process scheduler even more important.

And third, our results show that even equal thread distribution decreases the best-to-worst-assignment performance gap, the performance difference we measure is still notable, up to 17% in the worst case. This means that the job scheduler has to be aware of the application features in order to make the optimal process co-schedule.

6. RELATED WORK

To our knowledge few characterization works have been done on a real massive multithreaded systems. In [7] the authors classify applications regarding their usage of shared processor resources by co-scheduling them with *base vectors* (i.e. micro-benchmarks that specifically stress a single CPU resource) and then measuring the slowdown that each *base vector* causes to the application and the slowdown that the *base vector* sustains. Tseng et al. [10] evaluate the scalability of a set of widely-spread commercial workloads and benchmarks. Their work is focused on measuring the gains of a massive SMT architecture in real business applications, and also make a brief analysis of the memory subsystem behavior. Both works are done using a UltraSPARC T1 processor.

In [16] the authors propose to use microarchitectural level performance information as feedback to decide which threads to schedule in a SMT processor. The SOS scheduler that use profile-based information to compose the workload is presented in [22]. SOS runs several application mixes (workloads), examines their performance and applies heuristics to identify optimal schedules. The workloads that present the best symbiosis among combined tasks are selected. Other works, though, propose techniques to co-schedule threads

that exhibit a good symbiosis in the shared cache levels and solve problems of cache contention [12, 6, 9].

In all these works, 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). That is, let's assume the workload composed of threads {A,B,C,D} assigned to a given core is identified as the optimal schedule. The performance is independent of how the threads are scheduled (e.g. {A,B,C,D}, {B,C,D,A}, {D,A,C,B}).

Kumar et al. [14] and Shelepov et al. [19] take on the complex task of scheduling in heterogeneous cores architectures using simulators and real systems, respectively. Although these architectures are different than the UltraSPARC T2, there are similarities between the different levels of resource sharing and the heterogeneous domains. Their scheduling proposals are aware of the characteristics of each heterogeneous core and, thus, they can exploit the variations in thread-level parallelism as well as inter- and intra-thread diversity. However, heterogeneous processors show only sharing at the highest levels of the cache (inter-core resource).

7. CONCLUSIONS

In this paper, we make a complete characterization of the resource sharing levels of the UltraSPARC T2 processor, a processor with three layers of hardware resource sharing. From the results presented in the study, we extract implications that affect OS design:

(1) The execution of a process that dominantly stresses IntraPipe or IntraCore processor resources is independent of the co-schedule of other processes running on remote cores. This dramatically reduces the complexity of the OS job scheduler: instead of analyzing all tasks running on the processor, the job scheduler can reduce the scope of the analysis to only tasks executing on the same core.

(2) We measure a significant slowdown because of the collision in globally-shared resources that depends on the number and characteristics of the concurrently running tasks (workload). We argue it is very important that when the job scheduler selects a workload, it takes into account the interference the selected tasks experience in the globally-shared resources. However, once the workload is selected, tasks that dominantly stress globally-shared resources interfere the same way regardless of the way they are mapped to hardware contexts of the processor.

(3) We determine the most critical resources of UltraSPARC T2 processor: the IFU at IntraPipe level, non-pipelined floating point execution units at the IntraCore level, and L2 cache at the InterCore level. Applications using these resources at each level are the most sensitive to interference from other tasks. These conclusions can be

used by the OS job scheduler to define the optimal workload and to distribute sensitive processes among different hardware domains (hardware pipes and cores) in order to avoid interference among them.

Finally, through a case study with a real multithreaded network application, we show that a resource-sharing aware job scheduler can improve the performance of a naive scheduler by 55% and a load-balance aware scheduler by 17%. We also show that the absolute performance difference increases with the number of threads simultaneously running on the processor and with the number of processor cores applications use. Moreover, in the future, with more cores per processor, more hardware contexts per core, and more tasks concurrently running on the processor, the effect of the optimal process co-scheduling on system performance will increase, making resource-aware process schedulers even more important. Simultaneously, the complexity of finding a proper schedule of a given workload scales rapidly with the number of cores an application uses and the number of tasks to schedule. We expect that characterizations, like the one done in this paper, will help the designing of resource-aware job schedulers able to fully exploit the capabilities of the multithreaded processors with several levels of hardware resource sharing.

Acknowledgments

This work has been supported by the Ministry of Science and Technology of Spain under contracts TIN-2007-60625, the HiPEAC European Network of Excellence and a Collaboration Agreement between Sun Microsystems and BSC. The authors wish to thank the reviewers for their comments, Jochen Behrens and Aron Silvertson from Sun Microsystems, and Roberto Gioiosa from BSC for their technical support.

8. REFERENCES

- [1] *OpenSPARCTM T2 Core Microarchitecture Specification*, 2007.
- [2] *Netra Data Plane Software Suite 2.0 Update 2 Users Guide*, 2008.
- [3] J. Aas. Understanding the Linux 2.6.8.1 CPU Scheduler. *SGI, 2005.*, 2005.
- [4] A. Agarwal, B. H. Lim, D. Kranz, and J. Kubiawicz. APRIL: A processor architecture for multiprocessing. Technical Report MIT/LCS/TM-450, 1991.
- [5] P. Crowley, M. A. Franklin, H. Hadimioglu, and P. Z. Onufryk. *Network Processor Design: Issues and Practices*, volume 1. 2002.
- [6] C. Dhruba, G. Fei, K. Seongbeom, and S. Yan. Predicting inter-thread cache contention on a chip multi-processor architecture. In *11th HPCA*, 2005.
- [7] D. Doucette and A. Fedorova. Base vectors: A potential technique for microarchitectural classification of applications. In *Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*, 2007.
- [8] A. Fedorova. *Operating system scheduling for chip multithreaded processors*. PhD thesis, Cambridge, MA, USA, 2006.
- [9] A. Fedorova, M. Seltzer, and M. Smith. Improving performance isolation on chip multiprocessors via an operating systems scheduler. In *16th PACT*, 2007.
- [10] T. Jessica H., Y. Hao, N. Shailabh, D. Niteesh, F. Hubertus, P. Pratap, I. Hiroshi, and N. Toshio. Performance studies of commercial workloads on a multi-core system. In *IISWC '07*, 2007.
- [11] R. Halstead and T. Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. In *ISCA-15*, 1988.
- [12] J. Kihm, A. Settle, A. Janiszewski, and D. A. Connors. Understanding the impact of inter-thread cache interference on ilp in modern smt processors. *The Journal of Instruction Level Parallelism*, 7, 2005.
- [13] E. Kohler, J. Li, V. Paxson, and S. Shenker. Observed Structure of Addresses in IP Traffic. In *2nd ACM SIGCOMM Workshop on Internet measurement*, 2002.
- [14] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA Heterogenous Multi-Core Architectures for Multithreaded Workload Performance. In *31st ISCA*, 2004.
- [15] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. *SIGPLAN Not.*, 31, 1996.
- [16] S. Parekh, S. Eggers, H. Levy, and J. Lo. Thread-sensitive scheduling for smt processors. Technical report, University of Washington, Department of Computer Science & Engineering, 2000.
- [17] P. Radojković, V. Čakarević, J. Verdú, A. Pajuelo, R. Gioiosa, F. Cazorla, M. Nemirovsky, and M. Valero. Measuring Operating System Overhead on CMT Processors. In *SBAC-PAD '08*, 2008.
- [18] M. J. Serrano, R. Wood, and M. Nemirovsky. A study on multistreamed superscalar processors. Technical Report 93-05, University of California Santa Barbara, 1993.
- [19] D. Shelepov, J. C. S. Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar. HASS: A scheduler for heterogeneous multicore systems. In *ACM SIGOPS Operating Systems Review*, 2009.
- [20] T. Sherwood, G. Varghese, and B. Calder. A Pipelined Memory Architecture for High Throughput Network Processors. In *30th ISCA*, 2003.
- [21] B. Smith. Architecture and applications of the HEP multiprocessor computer system. *Fourth Symposium on Real Time Signal Processing*, 1981.
- [22] A. Snively, D. M. Tullsen, and G. Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *ACM SIGMETRICS*, 2002.
- [23] S. Storino, A. Aipperspach, J. Borckenhagen R. Eickemeyer, S. Kunkel, S. Levenstein, and G. Uhlmann. A commercial multithreaded RISC processor. In *45th International Solid-State Circuits Conference*, 1998.
- [24] L. A. Torrey, J. Coleman, and Barton P. Miller. A comparison of interactivity in the Linux 2.6 scheduler and an MLFQ scheduler. *Software Practice and Experience*, 37, 2007.
- [25] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *22nd ISCA*, 1995.
- [26] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernandez, and M. Valero. Analysis of system overhead on parallel computers. In *16th PACT*, 2007.