High-Performance Networking with Multi-Core and CMT Systems

www.world45.com

15th October 2007

Mariusz Nowostawski, Callum McKenzie and Andras Lipoth
World45 Ltd

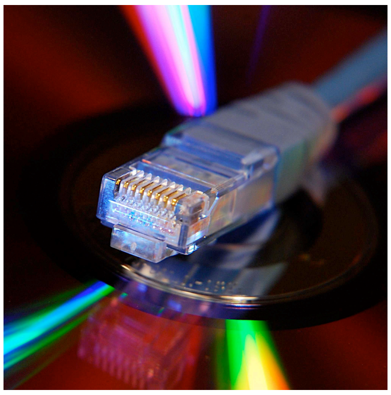
The recent expansion of mainstream computing hardware into the multi-core and chip-level multi-threading realm has produced a need for new software strategies. In parallel, the development of virtualization now allows us to write real-time code with dedicated hardware resources while retaining traditional operating systems in a neighboring domain. In this paper we discuss our work at World45 to take advantage of "bare-metal" domains and the techniques we use for high-speed inter-domain communication. We particularly focus on networking applications with relevance to ISPs, telecommunications companies and the VoIP industry.

In recent years technological advances in microprocessor architectures have reshaped the high performance and high throughput computational landscape[1]. The major focus has been multi-core CPUs and on chip-multi-threading (CMT) within individual cores. Two major themes have dominated the use of these computing resources.

The first is virtualization. The ability to partition computational resources into domains, each functioning as if it were a separate computer. This has been driven by the need to reduce energy consumption and rack-space footprint as well as assisting with centralized management. It also allows resources to be dynamically allocated to meet demand. Virtualization didn't originate with multi-core CPUs, but it has has been assisted by their rise - especially in their ability to dedicate cores to a domain allowing real-time guarantees. The appearance of IO-MMUs has also allowed the IO space to be partitioned, reducing reliance on service domains and virtual device drivers.

The other major theme is the rise of parallel programming to take full advantage of the available resources. In case of multi-core architectures, especially in the case when execution units are numerous, special techniques for parallelisation are needed. These techniques include, but are not limited to, software multi-threading, pipelining and pipeline matrices and speculative execution. Without these techniques software applications are not able to benefit from the underlying highly parallel execution hardware.

[1]Spracklen L. and Abrahams S.G., *"Chip multithreading: opportunities and challenges"*, HPCA-11, p. 248-252 (2005)



At World45 we are combining these two trends. We have developed a specialized offload system that allows us to write high-performance offload-engines that can run in their own virtual domain, making full use of multi-core CPUs, while retaining a traditional application running on an unmodified operating system. This model allows us to easily augment existing applications with high-performance high-throughput capabilities for those parts of the application that can significantly benefit from the underlying highly parallel multi-core architecture. We utilize memory mapping between the domains to speed up communication and minimize memory accesses by allowing an offload engine to write directly into the application's memory space. Our primary focus is high-performance and high-throughput networking for the benefit of clients in the telecommunications and network industries.

The Niagara Chip

While we work with the standard range Intel and AMD processors the system of choice for the work described in this paper is the Sun Microsystems Niagara range. The Niagara chips provide more hardware threads (32 threads on the Niagara T1 and 64 threads on the T2) than the Intel and AMD options, as well as an IO-MMU to split the PCI bus amongst domains. This last feature allows an offload engine direct access to the networking hardware. Sun's hypervisor and logical domains software allows us to partition the machine with any combination of CPUs and memory. Solaris and Linux can both be run within these domains. Alternatively the operating system can be replaced with a light-weight layer used to load the code and assign threads to functions. The Netra Data Plane Software Suite (Netra DPS) from Sun is an example of such a system.

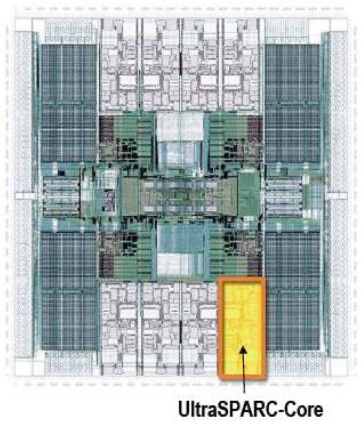


Figure 1: The UltraSPARC T1 chip with eight cores, each running four hardware threads. *Image courtesy of SUN Microsystems*

The hardware threads (or strands) on a T1 system are arranged into groups of four spread across eight cores. Within each core threads which are ready are executed in a round-robin manner. While this would seem to slow down an already slow execution rate, in practice the execution delay tends to match the L1 data cache latency and the hardware threads are rarely stalled waiting for data. The cores are connected by a cross-bar switch to 3 Mb of L2 cache which is in turn connected to DRAM via four 64-bit memory controllers working in parallel. Despite the width of the memory bus, main-memory

T1 Memory Access Latency	
L1	2 ns
L2	24 ns
DRAM	90 ns

latency is - as it always is for DRAM - a serious issue. So a lot of attention has to be paid to minimizing main-memory access.

Our Offload Engine

Our offload system spans two domains. A userspace library and kernel module run under Solaris (or, potentially, another OS) and a light-weight stub runs in an Netra DPS domain. Communication between the two domains is a combination of logical domain channels (LDC) - a packet based message-passing protocol run via the hypervisor - and shared memory. The shared memory is identical to shared memory in a tradition operating system, but run via the hypervisor and shares memory between domains. The Netra DPS part of the system runs all available hardware threads through a loop that waits for jobs to be transmitted from the OS. Once a thread starts executing a job, it will not be interrupted and can request other threads to assist it. The lack of pre-emption or operating system overhead allows real-time guarantees of performance and very-good repeatability of execution time. All hardware threads in the offload domain are available for execution, although it is always possible to request more work than there are threads - in this case new jobs are queued.

To demonstrate the scalability of our offload system we have made a comparison with OpenMP. OpenMP[2] is the traditional way to create high performance multi-threaded C programs on Unix. It is widely supported and scales well with small numbers of CPUs. Unfortunately, operating system overhead and scheduling issues can easily make it untenable for large numbers of CPUs. As can be seen to the right, with larger numbers of CPUs the run-time starts to increase and becomes highly variable. By the time 20 CPUs are utilized, performance has returned to a single CPU level. Conversely, our offload engine scales perfectly: the solid line is a power-law fit with an exponent of very close to -1. In other words, n threads reduce the run time by a factor of $1/n$. This example fits in the L2 cache and isn't limited by main-memory bandwidth; if we become bandwidth limited the execution time will flatten out. Shared memory also produces a drop in performance due to cache coherency issues - this is the primary reason that the OpenMP code is faster than the offload engine at low CPU numbers. The job-creation overhead is about 20 μs and is not

[2] www.openmp.org

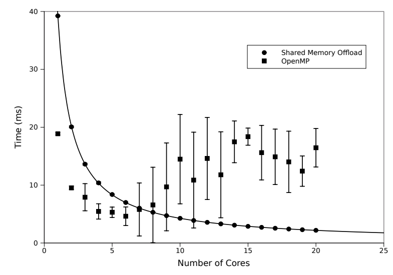


Figure 2: The perfect scaling of our offload system beats OpenMP with many threads.

significant in this example.

We will now consider some network applications. At the heart of this work is the realization that the bottle-neck for high-performance networking is not the network interface, but instead the calculation resources necessary to handle the large amounts of data available coming from a fully-utilized network interface. In a normal operating system, there are several layers of abstraction and this frequently leads to copying of data as it progresses up through the network stack. Main-memory latency (and even L2 cache latency) is significant when compared to the rate at which packets arrive[3] on even modest links and should be avoided at all costs. Operating systems also have other things to do as well as processing the packets and this makes maintaining a full speed 1 Gb/s ethernet link difficult. At 10 Gb/s this task becomes intractable with current operating system designs.

[3]At worst case (small packets and full speed) a 1 Gb/s ethernet link leaves only 672 ns to process a packet. A single main-memory access takes 90 ns.

Our solution is to move the processing of network traffic to the offload domain. This includes the device driver. This dedicates CPU and memory resources to the problem, while the IO-MMU allows direct access to the PCI bus and the network hardware. In the general case this is not particularly useful, due to a lack of universal strategies that work for all protocols. Instead we will look at optimizing for specific jobs, specifically packet classification and RDMA (Remote Direct Memory Access).

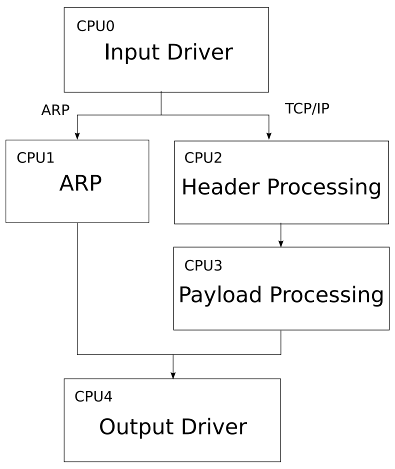


Figure 3: Pipelining and parallelizing a network stack.

The exact distribution of resources in the offload domain does, of course, depend on the application. A simplified example of how to segment a networking stack can be seen to the right. In this configuration infrequent operations such as ARP packet processing are split off into their own hardware thread to keep the main pipeline clean. The main pipeline - shown here as header and payload processing - can be broken up as necessary. With some protocols, e.g. RDMA, it is possible to parallelize the payload processing since the payloads are independent of each other. Other work-loads, e.g. intrusion detection systems (IDS), must view the data stream holistically and instead benefit from pipelining. In this last case, the IDS pipeline could be run in parallel with the main processing pipeline (assuming intervention can occur in a timely manner). Advanced networking hardware, e.g. Sun's "Neptune", provides hardware support for parallelizing the driver stage.

Remote Direct Memory Access

RDMA provides an interesting case: This protocol allows two computers to transfer data between main memories and is intended to be implemented by smart network cards with minimal CPU intervention. We take a dumb network card and dedicated CPU resources to emulate this in a more flexible manner.

The RDMA protocol gives us some help: Delivery of data to memory does not have to be sequential and each packet contains enough information to place it in memory. These features, intended to avoid re-ordering queues in network hardware allow us to parallelize the processing of packets without a final bottleneck as the stream is re-serialized. The only necessary synchronization is to determine when all the data has been received. Additionally the shared memory allows us to place the packet payload directly into userspace memory without additional copying.

Packet Classification

Another application we are exploring is packet classification. We are aiming to produce a software module that can match 10^5 rules on a 1 GB/s link. This module can then be parallelized to handle faster links. Within each module, some parallelism is possible, for example the matching of rules to source and destination fields is performed in parallel. The matches are then merged and the highest priority match picked in successive pipeline stages.

In this problem memory access latency is a significant delay since the rule tables are large and will not fit in L1 cache. Clever algorithms reduce the number of random lookups until we can complete the process with a series of predictable - and prefetchable - memory accesses.

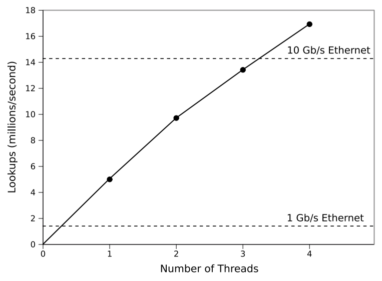


Figure 4: Efficiency and scaling in the front-end matcher of our packet classifier.

Summary

At World45 we are actively exploiting the opportunities provided by the combination of multiple-core CPUs and virtualization. Our offload-system allows us to provide specially-designed solutions for time and throughput critical applications while retaining the availability of traditional operating system services.