NIAGARA: A 32-WAY MULTITHREADED SPARC PROCESSOR

THE NIAGARA PROCESSOR IMPLEMENTS A THREAD-RICH ARCHITECTURE DESIGNED TO PROVIDE A HIGH-PERFORMANCE SOLUTION FOR COMMERCIAL SERVER APPLICATIONS. THE HARDWARE SUPPORTS 32 THREADS WITH A MEMORY SUBSYSTEM CONSISTING OF AN ON-BOARD CROSSBAR, LEVEL-2 CACHE, AND MEMORY CONTROLLERS FOR A HIGHLY INTEGRATED DESIGN THAT EXPLOITS THE THREAD-LEVEL PARALLELISM INHERENT TO SERVER APPLICATIONS, WHILE TARGETING LOW LEVELS OF POWER CONSUMPTION.

Poonacha Kongetira
Kathirgamar Aingaran
Kunle Olukotun
Sun Microsystems

•••••• Over the past two decades, microprocessor designers have focused on improving the performance of a single thread in a desktop processing environment by increasing frequencies and exploiting instruction level parallelism (ILP) using techniques such as multiple instruction issue, out-of-order issue, and aggressive branch prediction. The emphasis on single-thread performance has shown diminishing returns because of the limitations in terms of latency to main memory and the inherently low ILP of applications. This has led to an explosion in microprocessor design complexity and made power dissipation a major concern.

For these reasons, Sun Microsystems' Niagara processor takes a radically different approach to microprocessor design. Instead of focusing on the performance of single or dual threads, Sun optimized Niagara for multithreaded performance in a commercial server environment. This approach increases application performance by improving throughput, the total amount of work done across multiple threads of execution. This is especially effective in commercial server applications such as databases[1] and Web services,[2] which tend to have workloads with large amounts of thread level parallelism (TLP).

In this article, we present the Niagara processor's architecture. This is an entirely new implementation of the Sparc V9 architectural specification, which exploits large amounts of on-chip parallelism to provide high throughput. Niagara supports 32 hardware threads by combining ideas from chip multiprocessors[3] and fine-grained multithreading.[4] Other studies[5] have also indicated the significant performance gains possible using this approach on multithreaded workloads. The parallel execution of many threads effectively hides memory latency. However, having 32 threads places a heavy demand on the memory system to support high band-

Table 1. Commercial server applications.

Benchmark	Application category	Instruction-level parallelism	Thread-level parallelism	Working set	Data sharing
Web99	Web server	Low	High	Large	Low
JBB	Java application server	Low	High	Large	Medium
TPC-C	Transaction processing	Low	High	Large	High
SAP-2T	Enterprise resource planning	Medium	High	Medium	Medium
SAP-3T	Enterprise resource planning	Low	High	Large	High
TPC-H	Decision support system	High	High	Large	Medium

width. To provide this bandwidth, a crossbar interconnects scheme routes memory references to a banked on-chip level-2 cache that all threads share. Four independent on-chip memory controllers provide in excess of 20 Gbytes/s of bandwidth to memory.

Exploiting TLP also lets us improve performance significantly without pushing the envelope on CPU clock frequency. This and the sharing of CPU pipelines among multiple threads enable an area- and power-efficient design. Designers expect Niagara to dissipate about 60 W of power, making it very attractive for high compute density environments. In data centers, for example, power supply and air conditioning costs have become very significant. Data center racks often cannot hold a complete complement of servers because this would exceed the rack's power supply envelope.

We designed Niagara to run the Solaris operating system, and existing Solaris applications will run on Niagara systems without modification. To application software, a Niagara processor will appear as 32 discrete processors with the OS layer abstracting away the hardware sharing. Many multithreaded applications currently running on symmetric multiprocessor (SMP) systems should realize performance improvements. This is consistent with observations from previous multithreaded-processor development at Sun[6,7] and from Niagara chips and systems, which are undergoing bring-up testing in the laboratory.

Recently, the movement of many retail and business processes to the Web has triggered the increasing use of commercial server applications (Table 1). These server applications exhibit large degrees of client request-level parallelism, which servers using multiple threads can exploit. The key performance metric for a server running these applications is the sustained throughput of client requests.

Furthermore, the deployment of servers commonly takes place in high compute density installations such as data centers, where supplying power and dissipating server-generated heat are very significant factors in the center's cost of operating. Experience at Google shows a representative power density requirement of 400 to 700 W/sq. foot for racked server clusters.[2] This far exceeds the typical power densities of 70 to 150 W/foot^2 supported by commercial data centers. It is possible to reduce power consumption by simply running the ILP processors in server clusters at lower clock frequencies, but the proportional loss in performance makes this less desirable. This situation motivates the requirement for commercial servers to improve performance per watt. These requirements have not been efficiently met using machines optimized for single-thread performance.

Commercial server applications tend to have low ILP because they have large working sets and poor locality of reference on memory access; both contribute to high cache-miss rates. In addition, data-dependent branches are difficult to predict, so the processor must discard work done on the wrong path. Load-load dependencies are also present, and are not detectable in hardware at issue time, resulting in discarded work. The combination of low available ILP and high cache-miss rates causes memory access time to limit performance. Therefore, the performance advantage of using a complex ILP processor over a single-issue processor is not significant, while the ILP processor incurs the costs of high power and complexity, as Figure 1 shows.

However, server applications tend to have large amounts of TLP. Therefore, shared-memory machines with discrete single-threaded processors and coherent interconnect have tended to perform well because they exploit TLP. However, the use of an SMP composed of multiple processors designed to exploit ILP is neither power efficient nor cost-efficient. A more efficient approach is to build a machine using simple cores aggregated on a single die, with a shared on-chip cache and high bandwidth to large off-chip memory, thereby aggregating an SMP server on a chip. This has the added benefit of low-latency communication between the cores for efficient data sharing in commercial server applications.

Niagara overview

The Niagara approach to increasing throughput on commercial server applications involves a dramatic increase in the number of threads supported on the processor and a memory subsystem scaled for higher bandwidths. Niagara supports 32 threads of execution in hardware. The architecture organizes four threads into a thread group; the group shares a processing pipeline, referred to as the *Sparc pipe*. Niagara uses eight such thread groups, resulting in 32 threads on the CPU. Each SPARC pipe contains level-1 caches for instructions and data. The hardware hides memory and pipeline stalls on a given thread by scheduling the other threads in the group onto the SPARC pipe with a zero cycle switch penalty. Figure 1 schematically shows how reusing the shared processing pipeline results in higher throughput.

The 32 threads share a 3-Mbyte level-2 cache. This cache is 4-way banked and pipelined for bandwidth; it is 12-way set-associative to minimize conflict misses from the many threads. Commercial server code has data sharing, which can lead to high coherence miss rates. In conventional SMP systems using discrete processors with coherent system interconnects, coherence misses go out over low-frequency off-chip buses or links, and can have high latencies. The Niagara design with its shared on-chip cache eliminates these misses and replaces them with low-latency shared-cache communication.

The crossbar interconnect provides the communication link between Sparc pipes, L2 cache banks, and other shared resources on the CPU; it provides more than 200 Gbytes/s of bandwidth. A two-entry queue is available for each source-destination pair, and it can queue up to 96 transactions each way in the crossbar. The crossbar also provides a port for communication with the I/O subsystem. Arbitration for destination ports uses a simple age-based priority scheme that ensures fair scheduling across all requestors. The crossbar is also the point of memory ordering for the machine.

The memory interface is four channels of dual-data rate 2 (DDR2) DRAM, supporting a maximum bandwidth in excess of 20 Gbytes/s, and a capacity of up to 128 Gbytes. Figure 2 shows a block diagram of the Niagara processor.

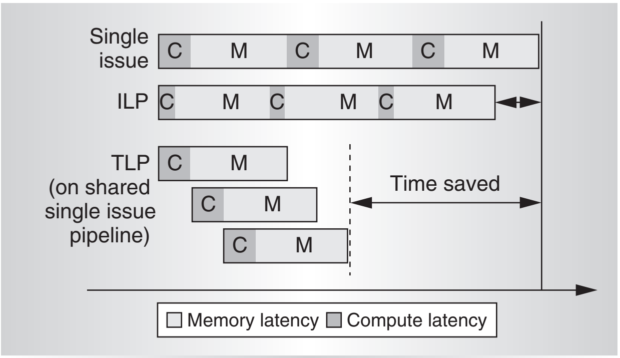

Figure 1. Behavior of processors optimized for TLP and ILP on commercial server workloads. In comparison to the single-issue machine, the ILP processor mainly reduces compute time, so memory access time dominates application performance. In the TLP case, multiple threads share a single-issue pipeline, and overlapped execution of these threads results in higher performance for a multithreaded application.

Sparc pipeline

Here we describe the Sparc pipe implementation, which supports four threads. Each thread has a unique set of registers and instruction and store buffers. The thread group shares the L1 caches, translation look-aside buffers (TLBs), execution units, and most pipeline registers. We implemented a single-issue pipeline with six stages (fetch, thread select, decode, execute, memory, and write back).

In the fetch stage, the instruction cache and instruction TLB (ITLB) are accessed. The following stage completes the cache access by selecting the way. The critical path is set by the 64-entry, fully associative ITLB access. A thread-select multiplexer determines which of

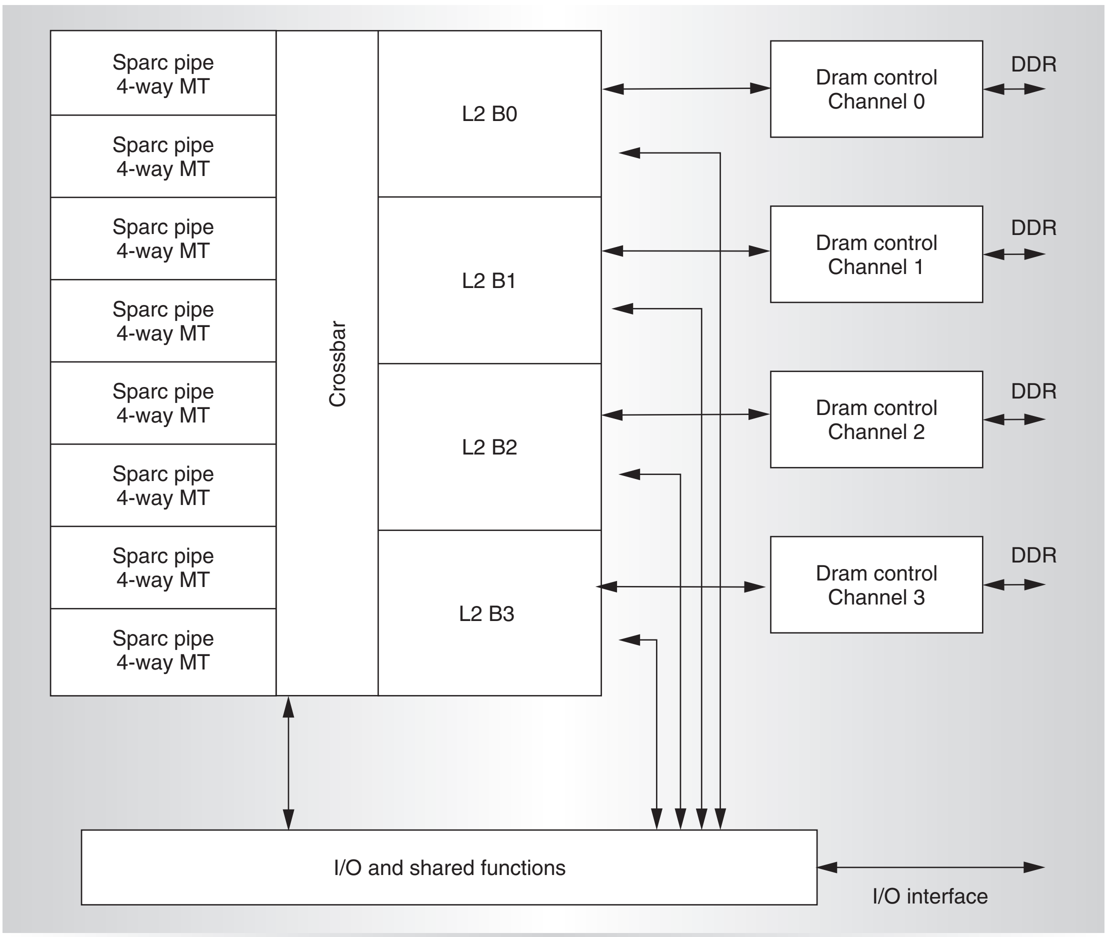

Figure 2. Niagara block diagram.

the four thread program counters (PC) should perform the access. The pipeline fetches two instructions each cycle. A predecode bit in the cache indicates long-latency instructions.

In the thread-select stage, the thread-select multiplexer chooses a thread from the available pool to issue an instruction into the downstream stages. This stage also maintains the instruction buffers. Instructions fetched from the cache in the fetch stage can be inserted into the instruction buffer for that thread if the downstream stages are not available. Pipeline registers for the first two stages are replicated for each thread.

Instructions from the selected thread go into the decode stage, which performs instruction decode and register file access. The supported execution units include an arithmetic logic unit (ALU), shifter, multiplier, and a divider. A bypass unit handles instruction results that must be passed to dependent instructions before the register file is updated. ALU and shift instructions have single-cycle latency and generate results in the execute stage. Multiply and divide operations are long latency and cause a thread switch.

The load store unit contains the data TLB (DTLB), data cache, and store buffers. The DTLB and data cache access take place in the memory stage. Like the fetch stage, the critical path is set by the 64-entry, fully associative DTLB access. The load-store unit contains four 8-entry store buffers, one per thread. Checking the physical tags in the store buffer can indicate read after write (RAW) hazards between loads and stores. The store buffer supports the bypassing of data to a load to resolve RAW hazards. The store buffer tag check happens after the TLB access in the early part of write back stage. Load data is available for bypass to dependent instructions late in the write back stage. Single-cycle

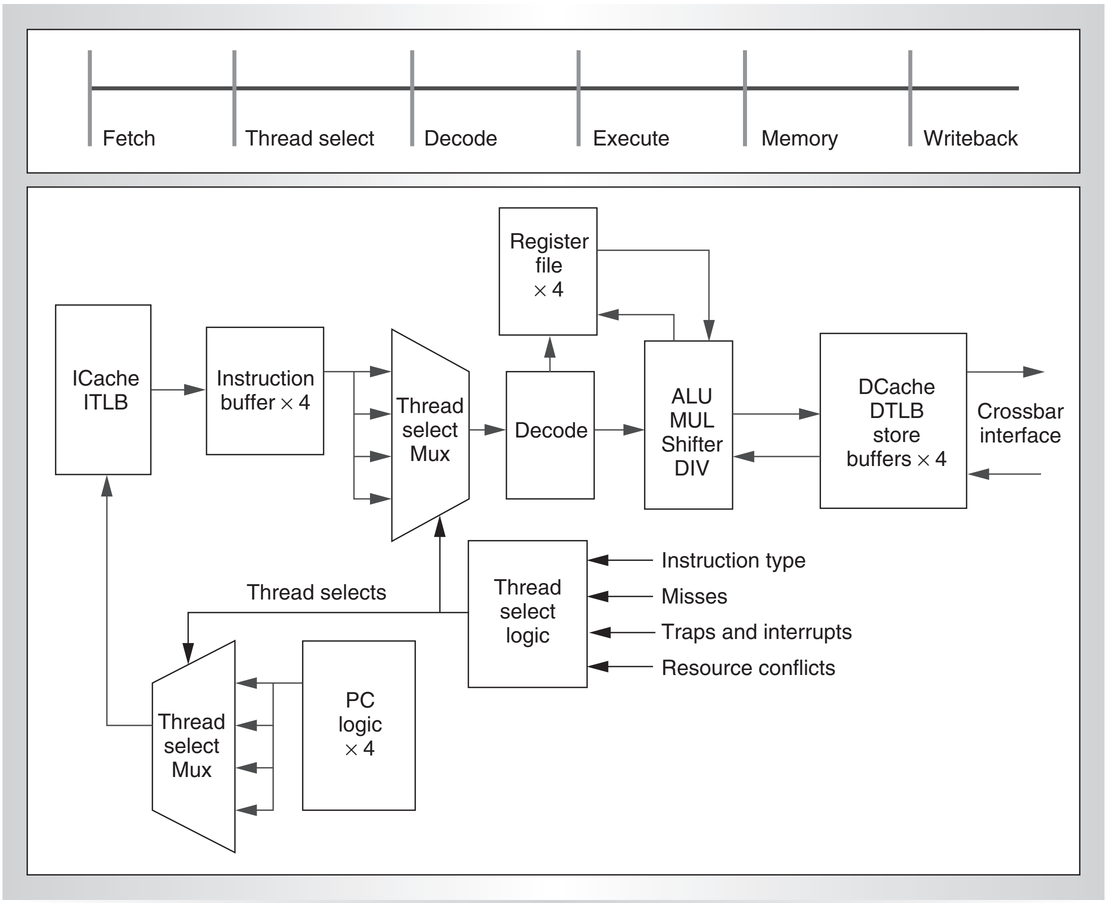

Figure 3. Sparc pipeline block diagram. Four threads share a six-stage single-issue pipeline with local instruction and data caches. Communication with the rest of the machine occurs through the crossbar interface.

instructions such as ADD will update the register file in the write back stage.

The thread-select logic decides which thread is active in a given cycle in the fetch and thread-select stages. As Figure 3 shows, the thread-select signals are common to fetch and thread-select stages. Therefore, if the thread-select stage chooses a thread to issue an instruction to the decode stage, the F stage also selects the same instruction to access the cache. The thread-select logic uses information from various pipeline stages to decide when to select or deselect a thread. For example, the thread-select stage can determine instruction type using a predecode bit in the instruction cache, while some traps are only detectable in later pipeline stages. Therefore, instruction type can cause deselection of a thread in the thread-select stage, while a late trap detected in the memory stage needs to flush all younger instructions from the thread and deselect itself during trap processing.

Pipeline interlocks and scheduling

For single-cycle instructions such as ADD, Niagara implements full bypassing to younger instructions from the same thread to resolve RAW dependencies. As mentioned before, load instructions have a three-cycle latency before the results of the load are visible to the next instruction. Such long-latency instructions can cause pipeline hazards; resolving them requires stalling the corresponding thread until the hazard clears. So, in the case of a load, the next instruction from the same thread waits for two cycles for the hazards to clear.

In a multithreaded pipeline, threads competing for shared resources also encounter structural hazards. Resources such as the ALU that have a one-instruction-per-cycle throughput present no hazards, but the divider, which has a throughput of less than one instruction per cycle, presents a scheduling problem. In this case, any thread that must execute a DIV

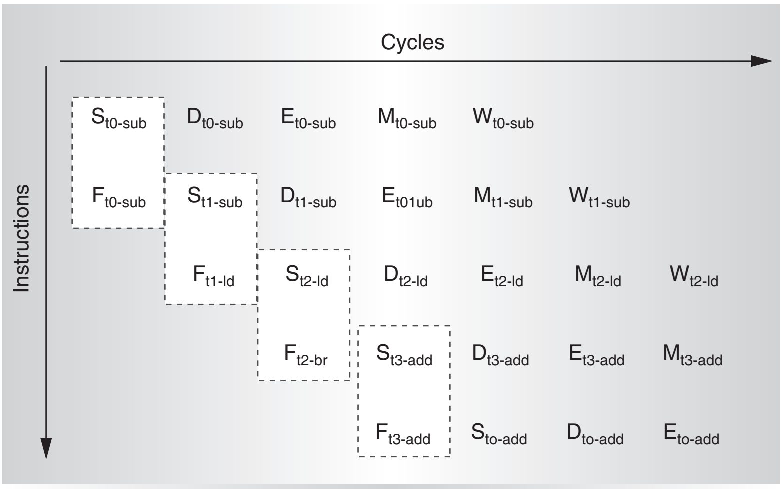
Cycles
Instructions
$S_{t0\text{-}sub}$ $D_{t0\text{-}sub}$ $E_{t0\text{-}sub}$ $M_{t0\text{-}sub}$ $W_{t0\text{-}sub}$
$F_{t0\text{-}sub}$ $S_{t1\text{-}sub}$ $D_{t1\text{-}sub}$ E_{t01ub} $M_{t1\text{-}sub}$ $W_{t1\text{-}sub}$
$F_{t1\text{-}ld}$ $S_{t2\text{-}ld}$ $D_{t2\text{-}ld}$ $E_{t2\text{-}ld}$ $M_{t2\text{-}ld}$ $W_{t2\text{-}ld}$
$F_{t2\text{-}br}$ $S_{t3\text{-}add}$ $D_{t3\text{-}add}$ $E_{t3\text{-}add}$ $M_{t3\text{-}add}$
$F_{t3\text{-}add}$ $S_{to\text{-}add}$ $D_{to\text{-}add}$ $E_{to\text{-}add}$

Figure 4. Thread selection: all threads available.

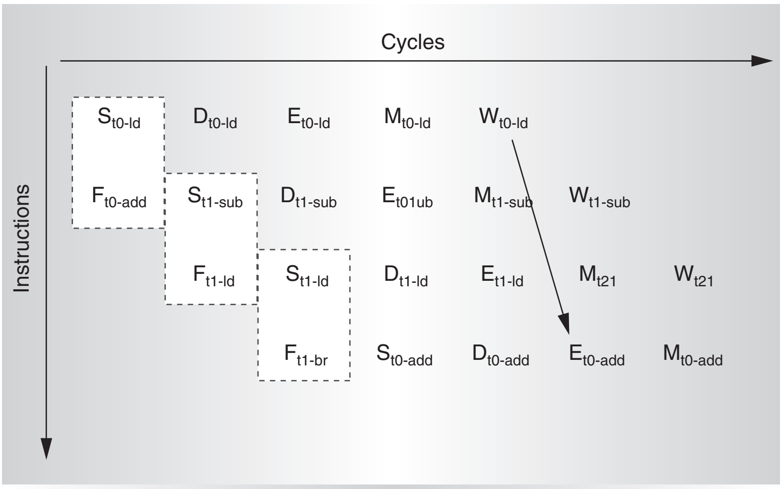
Cycles
Instructions
$S_{t0\text{-}ld}$ $D_{t0\text{-}ld}$ $E_{t0\text{-}ld}$ $M_{t0\text{-}ld}$ $W_{t0\text{-}ld}$
$F_{t0\text{-}add}$ $S_{t1\text{-}sub}$ $D_{t1\text{-}sub}$ E_{t01ub} $M_{t1\text{-}sub}$ $W_{t1\text{-}sub}$
$F_{t1\text{-}ld}$ $S_{t1\text{-}ld}$ $D_{t1\text{-}ld}$ $E_{t1\text{-}ld}$ M_{t21} W_{t21}
$F_{t1\text{-}br}$ $S_{t0\text{-}add}$ $D_{t0\text{-}add}$ $E_{t0\text{-}add}$ $M_{t0\text{-}add}$

Figure 5. Thread selection: only two threads available. The ADD instruction from *thread0* is speculatively switched into the pipeline before the hit/miss for the load instruction has been determined.

instruction has to wait until the divider is free. The thread scheduler guarantees fairness in accessing the divider by giving priority to the least recently executed thread. Although the divider is in use, other threads can use free resources such as the ALU, load-store unit, and so on.

Thread selection policy

The thread selection policy is to switch between available threads every cycle, giving priority to the least recently used thread. Threads can become unavailable because of long-latency instructions such as loads, branches, and multiply and divide. They also become unavailable because of pipeline stalls such as cache misses, traps, and resource conflicts. The thread scheduler assumes that loads are cache hits, and can therefore issue a dependent instruction from the same thread speculatively. However, such a speculative thread is assigned a lower priority for instruction issue as compared to a thread that can issue a nonspeculative instruction.

Figure 4 indicates the operation when all threads are available. In the figure, reading from left to right indicates the progress of an instruction through the pipeline. Reading from top to bottom indicates new instructions fetched into the pipe from the instruction cache. The notation $S_{t0\text{-}sub}$ refers to a Subtract instruction from thread 0 in the S stage of the pipe. In the example, the t0-sub is issued down the pipe. As the other three threads become available, the thread state machine selects *thread1* and deselects *thread0*. In the second cycle, similarly, the pipeline executes the t1-sub and selects t2-ld (load instruction from thread 2) for issue in the following cycle. When t3-add is in the *S* stage, all threads have been executed, and for the next cycle the pipeline selects the least recently used thread, *thread0*. When the thread-select stage chooses a thread for execution, the fetch stage chooses the same thread for instuction cache access.

Figure 5 indicates the operation when only two threads are available. Here *thread0* and *thread1* are available, while *thread2* and *thread3* are not. The t0-ld in the thread-select stage in the example is a long-latency operation. Therefore it causes the deselection of *thread0*. The *t0-ld* itself, however, issues down the pipe. In the second cycle, since *thread1* is available, the thread scheduler switches it in. At this time, there are no other threads available and the t1-sub is a single-cycle operation, so *thread1* continues to be selected for the next cycle. The subsequent instruction is a t1-ld and causes the deselection of *thread1* for the fourth cycle. At this time only *thread0* is speculatively available and therefore can be selected. If the first t0-ld was a hit, data can bypass to the dependent t0-add in the execute stage. If the load missed, the pipeline flushes the subsequent t0-add to the thread-select stage instruction buffer, and the instruction reissues when the load returns from the L2 cache.

Integer register file

The integer register file has three read and two write ports. The read ports correspond to those of a single-issue machine; the third read port handles stores and a few other three source instructions. The two write ports handle single-cycle and long-latency operations. Long-latency operations from various threads within the group (load, multiply, and divide) can generate results simultaneously. These instructions will arbitrate for the long-latency port of the register file for write backs. Sparc V9 architecture specifies the register window implementation shown in Figure 6. A single window consists of eight *in, local,* and *out* registers; they are all visible to a thread at a given time. The *out* registers of a window are addressed as the *in* registers of the next sequential window, but are the same physical registers. Each thread has eight register windows. Four such register sets support each of the four threads, which do not share register space among themselves. The register file consists of a total of 640 64-bit registers and is a 5.7 Kbyte structure. Supporting the many multiported registers can be a major challenge from both area and access time considerations. We have chosen an innovative implementation to maintain a compact footprint and a single-cycle access time.

Procedure calls can request a new window, in which case the visible window slides up, with the old outputs becoming the new inputs. Return from a call causes the window to slide down. We take advantage of this characteristic to implement a compact register file. The set of registers visible to a thread is the *working set*, and we implement it using fast register file cells. The complete set of registers is the *architectural set*; we implement it using compact six-transistor SRAM cells. A transfer port links the two register sets. A window changing event triggers deselection of the thread and the transfer of data between the old and new windows. Depending on the event type, the data transfer takes one or two cycles. When the transfer is complete, the thread can be selected again. This data transfer is independent of operations to registers from other threads, therefore operations on the register file from other threads can continue. In addition, the registers of all threads share the read circuitry because only one thread can read the register file in a given cycle.

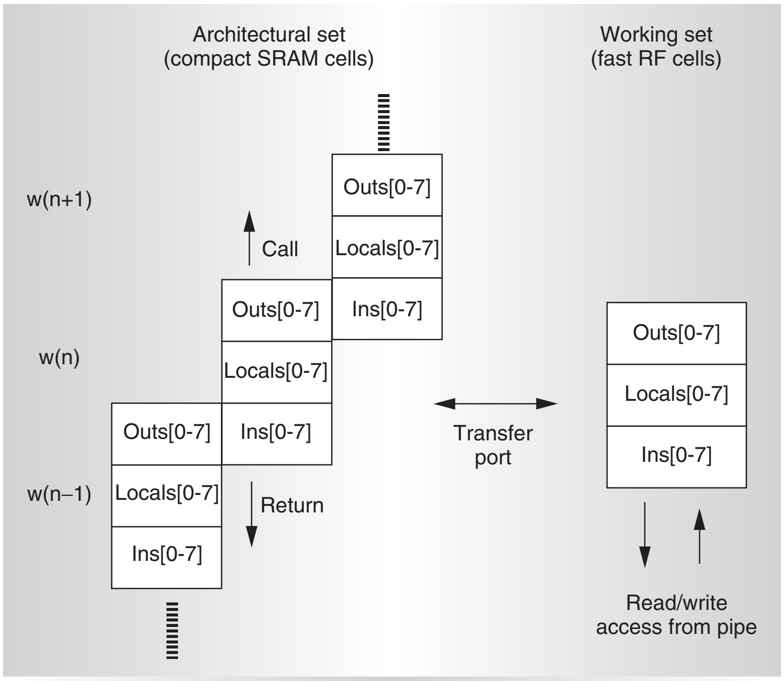

Figure 6. Windowed register file. *W*(*n* - 1), *w*(*n*), and *w*(*n* + 1) are neighboring register windows. Pipeline accesses occur in the working set while window changing events cause data transfer between the working set and old or new architectural windows. This structure is replicated for each of four threads,

Memory subsystem

The L1 instruction cache is 16 Kbyte, 4-way set-associative with a block size of 32 bytes. We implement a random replacement scheme for area savings, without incurring significant performance cost. The instruction cache fetches two instructions each cycle. If the second instruction is useful, the instruction cache has a free slot, which the pipeline can use to handle a line fill without stalling. The L1 data cache is 8 Kbytes, 4-way set-associative with a line size of 16 bytes, and implements a write-through policy. Even though the L1 caches are small, they significantly reduce the average memory access time per thread with miss rates in the range of 10 percent. Because commercial server applications tend to have large working sets, the L1 caches must be much larger to achieve significantly lower miss rates, so this sort of tradeoff is not favorable for area. However, the four threads in a thread group effectively hide the latencies from L1 and L2 misses. Therefore,

Hardware multithreading primer

A multithreaded processor is one that allows more than one thread of execution to exist on the CPU at the same time. To software, a dual-threaded processor looks like two distinct CPUs, and the operating system takes advantage of this by scheduling two threads of execution on it. In most cases, though, the threads share CPU pipeline resources, and hardware uses various means to manage the allocation of resources to these threads.

The industry uses several terms to describe the variants of multithreading implemented in hardware; we show some in Figure A.

In a *single-issue, single-thread machine* (included here for reference), hardware does not control thread scheduling on the pipeline. Single-thread machines support a single context in hardware; therefore a thread switch by the operating system incurs the overhead of saving and retrieving thread states from memory.

Coarse-grained multithreading, or *switch-on-event multithreading*, is when a thread has full use of the CPU resource until a long-latency event such as miss to DRAM occurs, in which case the CPU switches to another thread. Typically, this implementation has a context switch cost associated with it, and thread switches occur when event latency exceeds a specified threshold.

Fine grained multithreading is sometimes called *interleaved multithreading*; in it, thread selection typically happens at a cycle boundary. The selection policy can be simply to allocate resources to a thread on a round-robin basis with threads becoming unavailable for longer periods of time on events like memory misses.

The preceding types time slice the processor resources, so these implementations are also called time-sliced or vertical multithreaded. An approach that schedules instructions from different threads on different functional units at the same time is called *simultaneous multithreading* (SMT) or, alternately, *horizontal multithreading.* SMT typically works on superscalar processors, which have hardware for scheduling across multiple pipes.

Another type of implementation is *chip multiprocessing*, which simply calls for instantiating single-threaded processor cores on a die, perhaps sharing a next-level cache and system interfaces. Here, each processor core executes instructions from a thread independently of the other thread, and they interact through shared memory. This has been an attractive option for chip designers because of the availability of cores from earlier processor generations, which, when shrunk down to present-day process technologies, are small enough for aggregation onto a single die.

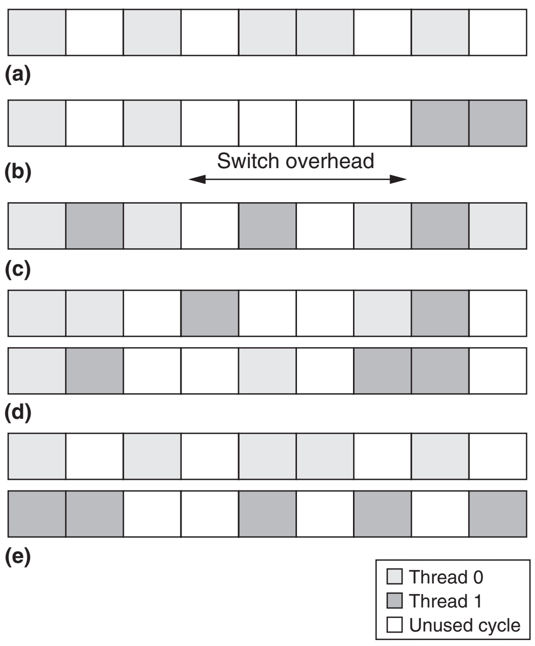

Figure A. Variants of multithreading implementation hardware: single-issue, single-thread pipeline (a); single-issue, 2 threads, coarse-grain threading (b); single-issue, 2 threads, fine-grain threading (c); dual-issue, 2 threads, simultaneous threading (d); and single-issue, 2 threads, chip multiprocessing (e).

the cache sizes are a good trade-off between miss rates, area, and the ability of other threads in the group to hide latency.

Niagara uses a simple cache coherence protocol. The L1 caches are write through, with allocate on load and no-allocate on stores. L1 lines are either in valid or invalid states. The L2 cache maintains a directory that shadows the L1 tags. The L2 cache also interleaves data across banks at a 64-byte granularity. A load that missed in an L1 cache (load miss) is delivered to the source bank of the L2 cache along with its replacement way from the L1 cache. There, the load miss address is entered in the corresponding L1 tag location of the directory, the L2 cache is accessed to get the missing line and data is then returned to the L1 cache. The directory thus maintains a sharers list at L1-line granularity. A subsequent store from a different or same L1 cache will look up the directory and queue up invalidates to the L1 caches that have the line. Stores do not update the local caches until they have updated the L2 cache. During this time, the store can pass data to the same thread but not to other threads; therefore, a store attains global visibility in the L2 cache. The crossbar establishes memory order between transactions from the same and different L2 banks, and guarantees delivery of transactions to L1 caches in the same order. The L2 cache follows a copy-back policy, writing back dirty evicts and dropping clean evicts. Direct memory access from I/O devices are ordered through the L2 cache. Four channels of DDR2 DRAM provide in excess of 20 Gbytes/s of memory bandwidth.

Niagara systems are presently undergoing testing and bring up. We have run existing multithreaded application software written for Solaris without modification on laboratory systems. The simple pipeline requires no special compiler optimizations for instruction scheduling. On real commercial applications, we have observed the performance in lab systems to scale almost linearly with the number of threads, demonstrating that there are no bandwidth bottlenecks in the design. The Niagara processor represents the first in a line of microprocessors that Sun designed with many hardware threads to provide high throughput and high performance per watt on commercial server applications. The availability of a thread-rich architecture opens up new avenues for developers to efficiently increase application performance. This architecture is a paradigm shift in the way that microprocessors have been designed thus far. MICRO

Acknowledgments

This article represents the work of the very talented and dedicated Niagara development team, and we are privileged to present their work.

References

1. S.R. Kunkel et al., "A Performance Methodology for Commercial Servers," *IBM J. Research and Development,* vol. 44, no. 6, 2000, pp. 851-872.
2. L. Barroso, J. Dean, and U. Hoezle, "Web Search for a Planet: The Architecture of the Google Cluster," *IEEE Micro,* vol 23, no. 2, Mar.-Apr. 2003, pp. 22-28.
3. K. Olukotun et al., "The Case for a Single Chip Multiprocessor," *Proc. 7th Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS VII),* 1996, pp. 2-11.
4. J. Laudon, A. Gupta, and M. Horowitz, "Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations," *Proc. 6th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS VI), ACM Press, 1994, pp. 308-316.
5. L. Barroso et al., "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing," *Proc. 27th Ann. Int'l Symp. Computer Architecture* (ISCA 00), IEEE CS Press, 2000, pp. 282-293.
6. S. Kapil, H. McGhan, and J. Lawrendra, "A Chip Multithreaded Processor for Network-Facing Workloads," *IEEE Micro,* vol. 24, no. 2, Mar.-Apr. 2004, pp. 20-30.
7. J. Hart et al., "Implementation of a 4th-Generation 1.8 GHz Dual Core Sparc V9 Microprocessor," *Proc. Int'l Solid-State Circuits Conf.* (ISSCC 05), IEEE Press, 2005, http://www.isscc.org/isscc/2005/ap/ISSCC2005AdvanceProgram.pdf.

Poonacha Kongetira is a director of engineering at Sun Microsystems and was part of the development team for the Niagara processor. His research interests include computer architecture and methodologies for SoC design. Kongetira has a MS from Purdue University and BS from Birla Institute of Technology and Science, both in electrical engineering. He is a member of the IEEE.

Kathirgamar Aingaran is a senior staff engineer at Sun Microsystems and was part of the development team for Niagara. His research interests include architectures for low power and low design complexity. Aingaran has an MS from Stanford University and a BS from the University of Southern California, both in electrical engineering. He is a member of the IEEE.

Kunle Olukotun is an associate professor of electrical engineering and computer science at Stanford University. His research interests include computer architecture, parallel programming environments, and formal hardware verification. Olukotun has a PhD in computer science and engineering from the University of Michigan. He is a member of the IEEE and the ACM.

Direct questions and comments about this article to Poonacha Kongetira,MS: USUN05-215, 910 Hermosa Court, Sunnyvale, CA 94086; poonacha.kongetira@sun.com.

For further information on this or any other computing topic, visit our Digital Library at http://www.computer.org/publications/dlib.