Sun’s Niagara Processor
A Multi-Threaded & Multi-Core CPU

Kai Johnson and Matthew Rathbone

Table of Contents

Introduction ........................................................................................................................................... 2
Pipeline Architecture .......................................................................................................................... 2
  Overview .......................................................................................................................................... 2
  An Example ....................................................................................................................................... 3
    Description ...................................................................................................................................... 4
    Discussion ....................................................................................................................................... 6
Memory Architecture .......................................................................................................................... 6
Physical Architecture .......................................................................................................................... 7
  Overview .......................................................................................................................................... 7
  L1 Cache .......................................................................................................................................... 7
  L2 Cache .......................................................................................................................................... 7
  DRAM Controller ............................................................................................................................. 8
  The Crossbar ..................................................................................................................................... 8
  IO Devices ....................................................................................................................................... 9
  Discussion ....................................................................................................................................... 9
Cache Coherency and Memory Access Patterns .................................................................................. 10
  Overview ........................................................................................................................................ 10
  Walkthrough ................................................................................................................................... 10
  Walkthrough Summary .................................................................................................................. 14
  Discussion ....................................................................................................................................... 14
System Benchmarks .......................................................................................................................... 14
  Power ............................................................................................................................................ 15
  Heat ............................................................................................................................................... 15
  Performance ..................................................................................................................................... 16
Conclusion ......................................................................................................................................... 17
Our Work .......................................................................................................................................... 17
Primary Reference Points .................................................................................................................. 18
Introduction
Sun's UltraSPARC T1 (codenamed "Niagara") is a 32-way multithreaded Sparc processor intended for high-performance computing applications such as web or database servers. Its 32 native hardware threads allow it the luxury of zero cycle context-switching among threads. The memory architecture of the machine is arranged in a hierarchy to ensure scalability while at the same time enabling efficient sharing of data.\(^1\) By exploiting the high memory access rates of such server applications, the Niagara is able to use a simple 6-stage CPU running at a little over 1 Ghz to achieve excellent throughput with minimal power consumption. In this report we aim to review the major architecture decisions of the Niagara T1 processor, demonstrate the workings of vital components through examples, and provide some rough insights as to system performance.

Pipeline Architecture
The Niagara pipeline is straightforward when compared to many other architectures, such as the x86.\(^2\) The x86 includes several extra components, such as the branch target buffer which enables branch prediction, and the entire pre-fetch stage of the pipeline. The simpler architecture of the Niagara also draws less power, enabling the Niagara to operate efficiently with considerably lower electricity usage.\(^3\)

Overview
The SPARC pipeline has six stages:

1. fetch
2. thread select
3. decode
4. execute
5. memory
6. write back

The CPU works like many do, so basic details about fetching and decoding have been omitted. The T-series architecture is most interesting in the first two stages, instruction fetching and thread selection. The pipeline has a thread selection multiplexer that chooses which of the four threads' program counters will actually continue processing for each CPU cycle. It chooses from among the available threads for which instruction to issue to the downstream stages.

---

\(^1\) The claims of future scalability are, in fact, becoming reality - the planned UltraSPARC T3 will have 128 hardware threads with mostly the same memory architecture. http://www.theregister.co.uk/2009/09/11/sun_sparc_roadmap_revealed/

\(^2\) http://www.laynetworks.com/block%20diagram%20of%20the%20pentium.htm

As the figure above shows, the pipeline contains replicated instruction buffers, register files, data cache, and program counter logic units - one for each thread (indicated by the "x4" in the respective elements). As we will see in the step by step example, this allows the pipeline to context switch between two threads at the hardware level without a single cycle missed. This zero cycle context switching is of great value in memory-bound processing, since it allows the CPU to hide memory latency without the overhead of the standard, expensive context switching.

Once the pipeline has chosen a thread's instruction to execute for the current cycle, it passes it to the decode stage, which accesses registers if necessary and decodes the instruction. Instructions that use the arithmetic logic unit (ALU) or the shifter complete in a single cycle, leaving that thread ready for the next cycle's thread selection. A multiply or divide operation has longer latency, and so causes a thread switch for the next CPU cycle.

**An Example**

Suppose there are 3 threads on a single core of a Niagara machine. The threads are each doing their own work:
Below is an example scheduling graph that shows the progression of the threads' instructions down the pipeline. It is followed by a textual description.

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>S t1-</td>
<td>D t1-</td>
<td>E t1-</td>
<td>M t1-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>load A</td>
<td>load A</td>
<td>load A</td>
<td>cache</td>
<td>miss !</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F t1-</td>
<td>S t2-</td>
<td>D t2-</td>
<td>E t2-</td>
<td>M t2-</td>
<td>W t2-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>add</td>
<td>add</td>
<td>add</td>
<td>add</td>
<td>add</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F t2-</td>
<td>S t3-</td>
<td>D t3-</td>
<td>E t3-</td>
<td>M t3-</td>
<td>W t3-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>store B</td>
<td>sub</td>
<td>sub</td>
<td>sub</td>
<td>sub</td>
<td>sub</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F t3-</td>
<td>S t1-</td>
<td>D t1-</td>
<td>E t1-</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>add</td>
<td>add</td>
<td>add</td>
<td>(empty)</td>
<td>(empty)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F t1-</td>
<td>S t2-</td>
<td>D t2-</td>
<td>E t2-</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>load A</td>
<td>load B</td>
<td>load B</td>
<td>load B</td>
<td>load B</td>
<td>load B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F (empty)</td>
<td>S t3-</td>
<td>D t3-</td>
<td>E t3-</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add add</td>
<td>add add</td>
<td>add add</td>
<td>add add</td>
<td>add add</td>
<td>add add</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F t3-</td>
<td>S t3-</td>
<td>D t3-</td>
<td>E t3-</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mult</td>
<td>div</td>
<td>div</td>
<td>div</td>
<td>(cont'd)</td>
<td>(cont'd)</td>
<td>(cont'd)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>add</td>
<td>add</td>
<td>add</td>
<td>add</td>
<td>add</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F t3-</td>
<td>S t3-</td>
<td>D t3-</td>
<td>E t3-</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add add</td>
<td>add add</td>
<td>add add</td>
<td>add add</td>
<td>add add</td>
<td>add add</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Description**

The above chart, when read left to right, indicates an instruction's path through the pipeline. When read bottom to top in a column it indicates the work each stage of the pipeline is doing for the given cycle. Highlighted in red are the failed operations that cause empty pockets in the pipeline.
The fetch (F) and select (S) stages are fed by the same thread multiplexer, so in all cases, they will be operating on the same thread. In this case, the pipeline is primed with each thread's first instruction (so they are selected into the S stage, not the F stage). Let's assume the instruction cache and the data cache are empty. We are also only showing 3 threads on this processor - assume the fourth is absent.

Starting with cycle 1, thread 1 is loading memory address A. This is a long latency instruction, so the thread is deselected for the next instruction and the load continues down the pipeline. Thread 2 is selected with its simple add instruction. The following store is fetched for thread 2. Although the add instruction will complete in one cycle, the thread selector enforces fairness by using a least recently used selection algorithm. This means thread 3 will be selected next for the S and F stage. Thread 3’s subtraction instruction is selected and its next instruction (add) is fetched. By now, the CPU has executed 2 cycles since thread 1’s load, meaning thread 1 is now eligible for selection again (the pipeline optimistically assumes a cache hit).

Thread 1 selects it’s add, which operates on the load A from cycle 1. Because the load is still in the memory (M) stage of the pipeline, we don’t yet know if A was a cache hit. The add operation will need the results of the load in the Execution (E) stage, so the instruction can begin down the pipeline. In cycle 5, we discover that thread 1's load was indeed a miss! We will send out a request on the crossbar, which will end up at an L2 cache bank (see Memory Architecture section), which itself may need to go to DRAM before we get the value of A, so the thread’s execution must be stopped. The pipeline also needs to be cleared of any instructions upstream, so the add operation that was in the decode stage as of cycle 5 is cleared. This leaves an empty stage until cycle 9. Thread 1’s add cleared from the pipeline gets placed into the instruction cache until the load returns from L2, at which point thread 1 is ready and will be selected by the multiplexer next. Cycle 5 also reveals that thread 2 is done, so the fetch stage is empty. In a real system, a new thread would be scheduled into its place, but in our example, we are down to threads 1 and 3.

In cycle 6, we retire our first instruction - thread 2’s 2+1 is retired, and since there is no need for it to write back to a dependent upstream instruction, there is no further work to do. We select thread 3 because thread 1 is waiting on a cache miss. We issue the simple add down the pipeline.

Cycle 7 selects thread 3 again as the only ready thread available and issues a division down the pipeline while fetching the next instruction, another add. We also retire the first instruction from thread 3, the subtraction. Cycle 8 selects thread 3 again, and an add is issued down the pipeline. In cycle 9, the division instruction has hit the execute stage, and so must make use of the divider. This resource takes more than one cycle to complete, so the instruction stays in E for an extra cycle. Other instructions not using the divider may continue to make progress. In cycle 10, the divider is finishing the division while the ALU is adding thread 3’s next instruction. By cycle 11, both are done, but since the data cache the result is being written to is per-thread, only one thread can write to it per cycle. The division is written to
memory and finally, in cycle 12, the add operation enters the memory stage. We have left off the final stages of these instructions, but they would finish uneventfully.

Thread 1's access of the L2 cache takes on the order of hundreds of cycles to return. However when it does, the next instruction will be selected from the instruction cache, those following will be fetched from the program counter and execute relatively quickly, including the second load since memory address A will already be in the L1 cache.

**Discussion**
The important features illustrated above are:

- Hazard detection - loads force subsequent instructions on the same thread to wait for 2 cycles; use of divider or other multi-cycle execution resource may cause backup in instructions waiting for that resource
- Instruction cache holds instructions not yet ready to enter the pipeline (including for a long latency load, for instance)
- Zero cycle context switching
- Crossbar is only point of interaction with the rest of the entire system

Based on the sample code above, it is easy to see why zero cycle context switching is desirable - thread 1 begins with a load and, unless it is switched out with a high overhead cost, would force the CPU to sit there until the load returned and the next instruction could be issued (even with instruction-level parallelism, the CPU would sit idle since most of the following instructions depend on the memory loaded in the first instruction). Particularly with the cache miss seen above, the Sparc pipeline performs exceptionally well by scheduling the ready threads into selection while allowing other components of the system to service the pending request from waiting threads.

**Memory Architecture**
There are 5 core components to consider when describing the memory architecture of the Niagara T1 processor:

1. Sparc Pipelines (cores)
2. L1 Caches
3. L2 Caches
4. DRAM Controller
5. IO Devices (mostly out of the scope of this report)

We will consider these in relation to physical architecture, memory access patterns, and cache coherence.
Physical Architecture

![CPU Memory Overview](image)

Figure 2: CPU Memory Overview, taken from Sun documentation

Overview

- L1 cache is contained within each core and shared between the 4 threads in that core
- L2 cache is a ‘unified’ unit, in that any core can access any L2 bank
- A shared-region called the ‘crossbar’ serves as the connection between each core and the L2 cache banks
- Each channel of main memory is mapped to a single L2 bank, there is a 1-1 mapping
- To a cpu-core, the L2 cache represents main memory, there is no other way to access it
- IO devices are connected to both the crossbar and to L2 caches

L1 Cache

- Size: 16kb for instructions and 8kb for data, per core. There are 8 cores
- Blocks/Lines: block size for I-cache is 32 bytes, line size for D-cache is 16-bytes
- Associativity: both L1 caches are 4-way associative, so cache-lines from any of the L2 cache-banks can be mapped to any L1 cache line
- Replacement algorithm: random replacement
- States: there are two simple states: valid and invalid. Invalidation of cache lines is not handled at the L1 level (as we shall see later)
- Algorithms: write-through, with allocate-on-load and no-allocate-on-store

L2 Cache

- Size: 3mb banked across 4 L2 banks, interleaved on 64 byte boundaries
- Bank selection: based on physical address bits 7:6
- Latency: 23 cycles for a L1 data cache miss, 22 for a L1 instruction cache miss
- Line size: 64-bytes (as per the banking boundary)
- Associativity: this cache is 12-way set-associative
- Replacement algorithm: unknown

**DRAM Controller**
- Each DRAM controller maps to a single L2 Bank
- The DRAM controller is ON-CHIP
- 2 DDR2 DIMMs per memory controller
- The T1 can support up to 128GB, 32GB per controller
- Speed: ~20Gbytes/sec per memory controller

**The Crossbar**

![Crossbar Diagram](image)

*Figure 3: A simple representation of the crossbar from Sun documentation*

- **Job:** pass messages between the cores, the L2 cache, and the IO bus, it also maintains transaction order regardless of source
- **Structure:** switch (based on memory address)
- **Speed:** It runs at the core clock-frequency, providing > 200Gbytes/sec of bandwidth
- **Special properties:**
  - It is basically a set of high-speed switches
  - Non-blocking
  - Queue-like behavior (FIFO)
  - Ordering maintained across all cores
  - Allows multiple packets to be enqueued from each source at the same time
  - Technically it is more like several one directional crossbars, for example there is a crossbar for Core -> L2 transactions and a separate crossbar for L2 -> Core transactions
Requests from multiple cores are served at the same time, in parallel

**IO Devices**

- Requests for DMA are ordered by the L2 cache, and presumably DMA writes directly to the L2 cache, although I could not validate this assumption.

**Discussion**

There are four main points to highlight with respect to the Niagara memory architecture.

Firstly, because there are 4 on-chip memory controllers there is no need for an external chipset. Thus memory access is much faster than in a conventional x86 processor design as memory can be closer to the CPU and load is spread across 4 separate controllers. Sun hails this architecture by highlighting comparisons to Intel Xeon, Intel Itanium, and AMD Opteron processors, which have 6.4Gbytes/sec of memory bandwidth compared to the 20Gbytes/sec present on the T1. However Sun’s benchmarks also highlight that in practice the required memory bandwidth rarely exceeds 7Gbytes/sec, although there are artificial tests that support the system’s maximum memory bandwidth figures.

Secondly, having all eight cores share a single L2 cache is an important architectural difference because it eliminates the need for ‘snooping’. This structure also enables the hardware to present a single unified memory address-space to the operating system, something which is more complex (and a significant bottleneck) on systems which are more classically designed.

Thirdly, the structure of the crossbar enables the processor to make 8 simultaneous memory-addresses per cycle (one per core). This means that heavy CPU load does not constrain through-put and maintains the high memory access speeds for each core.

As with any system design, the T1’s memory architecture is not without its drawbacks. The clearest and most obvious issue is that creating multi-socket systems become a complicated technical challenge due to the combination of on-board memory controllers and the directory cache coherence policy. In other words with the T1 architecture you cannot simply upgrade your system by placing two T1 CPU’s on a dual socket motherboard to achieve 64 hardware threads. Sun has managed to overcome this issue with the T2+ which allows 4 64-thread processors per system, providing a total of 256 threads on one machine. In 2008 The Register also speculated that Sun is developing a 16-core, 16-thread Niagara KT processor along with an 8-socket motherboard capable of providing a total of 2,048 threads, although this prediction has yet to come to fruition.

A more subtle restriction is that due to the on-chip memory controllers, all DDR memory positions must contain an equal number of DIMMs, all of the same capacity, speed, and from the same manufacturer.

---

4 [http://www.theregister.co.uk/2008/06/23/sun_niagara_k2/](http://www.theregister.co.uk/2008/06/23/sun_niagara_k2/)
Cache Coherency and Memory Access Patterns

Overview
In a Niagara processor, cache-coherency is enabled by the L2 cache which keeps a directory of L1 cache-lines and controls the invalidation of those cache lines. Communication between cores and the L2 cache is done through the crossbar, which guarantees that transactions are delivered in the same order by which they are received. The L2 is the central point of control, and no changes are made to L1 cache lines without first updating the L2 cache.

Because all modifying transactions attain global visibility (through the shared L2 cache) there is no need for snooping, complex cache-line states, or coherency logic in the L1 cache.

Walkthrough
The best way to illustrate the memory access and coherence model is to walk through the following set of simple memory transactions:

1. Core1 issues a read for variable ‘A’
2. Core2 issues a read for variable ‘A’
3. Core1 writes a new value to variable ‘A’ (core 2 is doing something else, but does not evict ‘A’)
4. Core1 issues a read for variable ‘B’
5. Core1, thread 2 issues a read for the variable ‘B’
6. Core2 writes a new value to variable ‘B’

This will demonstrate the speed benefits of a single L2 cache, and also provide insight into how coherence is managed through a directory.

1. Core1 issues a read for variable ‘A’

(Above Left) A load is issued by core1 for the cache line containing ‘A’. This goes to the L2 cache. The L2 makes a directory entry linking Core1 with cache-line ‘A’. Notice how the request is automatically directed to the correct L2 cache bank (this is
based on memory address).

(Above Right) The correct cache line is fetched from memory, stored in the L2 cache, and forwarded to Core1.

2. Core2 Issues a read for variable ‘A’

(Above Left) Core 2 requests a copy of ‘A’, this request goes to the L2 cache as before. At this point, the L2 cache updates its directory so that it knows that both Core1 and Core2 hold cache-line ‘A’.

(Above Right) This time ‘A’ is already cached in L2, so it is returned directly, without requiring memory access.

3. Core1 writes a new value to variable ‘A’ (core 2 is doing something else, but does not evict ‘A’)

(Above Right)
(Top Left) Core1 writes-through the new value of ‘A’, it does not do anything to the version in the L1 cache (see discussion section).

(Top Right) On arrival at the L2 a directory lookup is performed to invalidate respective L1 cache lines.

(Bottom Left) The new value of ‘A’ is written to the L2 Cache.

(Bottom Right) Sometime in the future (not immediately), the new value of ‘A’ is flushed to memory.

4. Core 1 issues a read for variable ‘B’

(Above Left) As before, Core1’s cache doesn’t contain ‘B’, so a read-request is issued to the L2 cache, which makes an entry in its directory: ‘B’ -> Core1.

(Above Right) Again, the L2 cache reads in the value ln from memory and returns it to Core1.
5. Core 1, thread 2 issues a READ for variable ‘B’

(Above) As Core1 already has ‘B’ cached, it is loaded directly from the L1.

6. Core2 writes a new value for variable ‘B’
(Top Left) Core2 writes-through the new value of ‘B’. It does NOT have to load ‘B’ into its own L1 cache before doing so.

(Top Right) The cache directory is checked and Core1’s cache entry is invalidated.

(Bottom Left) The new value of ‘B’ is written to the L2 cache bank.

(Note) Any subsequent read of ‘B’ by Core1 will require reloading the value from the L2 cache.

Walkthrough Summary
- All threads on the same core can use values stored in that core’s L1 cache
- When writing a value to memory the core does not need to load the old value into its L1 cache prior to writing.
- Coherency and cache invalidations are centrally handled in the L2 cache
- We saw no snooping, this is important as it reduced a huge bottleneck and significantly simplifies both the cache coherency logic, and the number of transactions a core needs to issue

Discussion
The ease with which we can demonstrate a series of memory accesses says a great deal about the elegance of the cache-coherency algorithms of the Niagara. In multi-core x86 the cache coherency methodology was shoe-horned into an existing single-core model, whereas the ground-up design of the Niagara not only eases the conceptual load for the reader, but also removes a significant amount of processor complexity, improving the scalability of multiple threads.

There are still a couple of uncertain points surrounding the memory access patterns. Firstly, it is unclear what happens to L1 cache lines when a core issues a STORE transaction, it may invalidate its own cache on the way out, or it may not. It has been assumed that the latter is true due to the assertions in the documentation that “Stores do not update the local caches until they have updated the L2 cache”\(^5\). However, there are statements in the same and related documents that in part contradict this assumption. Secondly, although we can make reasonable assumptions, it is a little unclear as to how the cache-directory is maintained when L1 cache-line evictions occur. We know that an eviction must be prompted by a LOAD, so we assume that the L2 cache cross-references the evicted L1 cache-line in the cache-directory to remove existing entries. However, this has not been validated.

System Benchmarks
Sun promotes its line of Niagara server systems as the ultimate in high-throughput computing, ideal for most commercial server workloads because of three key system properties:

1. Low power consumption compared to offerings from other manufacturers
2. Lower operating temperatures

3. Significantly better performance for memory intensive threaded applications such as databases and web-servers

These three areas are of major concern to commercial data-center owners, as the costs of power consumption and cooling must be balanced with high-performing systems in order to ensure a profitable cost-performance ratio.

**Power**

Sun’s own power consumption statistics show a power consumption of 72w for a Niagara T1 processor running at 1.2ghz (with the fully loaded 32 threads across 8 cores) compared to 130w for a Pentium Extreme Edition running at 3.2ghz (with 4 threads across 2 cores). However there have been conflicting reports from online reviews which show the T1 running with similar power demands to a dual-cpu dual-core Opteron server.\(^6\) In fact the system actually ran with higher power demands, but the T1 server tested had more components than the Opteron.

**Heat**

The Niagara has a very different heat signature to a regular single core processor:

![Heat Map](image)

(Above Left) The heat-map of a traditional Single-Core processor.

(Above Right) The heat map of a Niagara T1 processor. These diagrams come from Sun Documentation.

This difference in heat distribution is believable given the larger die size, less complicated pipeline, and fairly even division of multiple fairly low-power cores. The benefit of the more spread-out heat pattern is three-fold: 1) cheaper cooling solutions may be used, 2) more T1 racks can be squeezed into a rack

\(^6\) http://don.blogs.smugmug.com/2006/08/15/sun-fire-coolthreads-t1000-review/
without overheating, and 3) the costs of temperature control for the same number of T1’s would also be significantly reduced.

Performance
According to SPEC (the standard performance evaluation corporation) web-server performance results from 2005, the Sun Niagara T1 processor (in a T1000 server) performs about twice as well as a Dell PowerEdge 2850 with two dual-core 2.8ghz Intel Xeon processors with both systems running java-servlet based web-servers.

This comparison was highlighted in the press around the time of the T1000/T2000 release\(^7\), but if we examine the cost of each of these servers, you can buy a refurbished Dell PE 2850 for around $400 (plus the cost of memory and storage), but a Sun T1000 costs in excess of $2000, so I am not sure if this is a fair benchmark.

Fortunately, we have the ability to view retrospective comparisons, and compare against a system such as the HP ProLiant DL380 G5, which costs about the same amount as a Sun T1000 on the second hand market.\(^8\) Disregarding the potential price inflation of the T100 due to scarcity, comparing these two similarly priced systems shows a different story. The HP system pulls in a SPECweb score of 20387, which is nearly twice the T1000 score of 10466, and still better than the T2000 score of 16407.\(^9\)

This lackluster performance is reflected in some online reviews such as that by SmugMug’s founder Don MacAskill.\(^10\) His performance review, although not an apples-to-apples comparison, reaches the conclusion that a traditional AMD Opteron solution (two dual-core 2.0ghz processors with 4gb of RAM) performs better in his benchmarks despite a 2.5x lower price point. He tests the setup using his photograph-serving web-server-application on Ubuntu Linux with upto 256 threads of execution.

However, it is not all bad-news for the Niagara. Colm MacAírthaigh posts some benchmark results for his ftp-site \(\text{ftp.heanet.ie}^{11}\). Interestingly he compares the Sun T2000 to the Dell 2850 mentioned earlier in this section. His report notes two areas of importance: 1) at the time, the cost of the T2000 was less than that of an x86 based server such as the Dell 2850, and 2) for throughput tests (such as requests-per second, and # of concurrent downloads) the T2000 outperforms the 2850 and a 1.5ghz Itanium by a significant factor.

\(^7\) For example, http://blogs.zdnet.com/Murphy/?p=524

\(^8\) Prices obtained from Google Product Search


\(^10\) http://don.blogs.smugmug.com/2006/08/15/sun-fire-coolthreads-t1000-review/

\(^11\) http://www.stdlib.net/~colmmacc/2006/03/23/niagara-vs-ftpheanetie-showdown/
Although online benchmark results seem mixed for the Niagara processor, there is some evidence in its favor. It has won awards for web-server performance\(^\text{12}\) and SPECweb 2005 results show it compares favorably against many other processors of the same generation. However it remains unclear as to whether the upfront-costs of purchasing a T1000 or T2000 system take it into the realms of better-performing high-end commercial servers where it no longer has a competitive edge.

**Conclusion**

The UltraSPARC T1 (and its successor, the T2) are exceptional machines in that they take the commercial server in a different direction than most multi-core processors. By recognizing memory latency as a major bottleneck in traditional CPU processing, Sun has designed a chip that copes with the relatively slow memory by allowing the CPU to continue processing other threads' instructions without missing a cycle. Allowing threads to run interleaved without context switching gives server applications running on the Niagara throughput as good as, or better than, many competitors, with a fraction of the power usage. The machine is not suited for CPU-bound computation, as its "time-sharing" of the pipeline provides sub-standard throughput in such a case. But in the case of memory- or I/O-bound workloads the Niagara is an excellent choice and given the projected power savings over its lifetime compared to competitors, it may be the most cost-effective choice as well.

**Our Work**

Our technical research for this report relied mostly on T1 and T2 discussion papers and technical overviews written by Sun and published in various industry journals. Of particular use was the 2005 IEEE paper "Niagara: A 32-Way Multithreaded Sparc Processor." While the step-by-step walkthroughs of the cache coherence structure and the pipeline design allowed us to dive into the details of the system, in a few cases these documents were insufficient to construct a full picture of the system. If we were to continue our work, a more thorough consultation of the developer manuals and a wider range of white-papers would hopefully provide the missing information we needed, and would also give us the opportunity to explore the Windowed Register File and its interaction with the rest of the system.

\(^{12}\) http://www.developers.net/sunicshowcase/view/2965
For performance analysis it would have been advantageous to gain access to an actual UltraSPARC T1 machine to perform simple benchmarking, but we have instead used the unofficial benchmarks of others to support our discussion.

**Primary Reference Points**

Unless otherwise mentioned, all information and images on the Sun Niagara processor came from one of the following documents:

   [http://wikis.sun.com/display/BluePrints/The+UltraSPARC+T1+Processor+-+High+Bandwidth+For+Throughput+Computing](http://wikis.sun.com/display/BluePrints/The+UltraSPARC+T1+Processor+-+High+Bandwidth+For+Throughput+Computing)
5. OpenSPARC T1 Microarchitecture Specification, Sun Microsystems, [http://www.opensparc.net/opensparc-t1/index.html](http://www.opensparc.net/opensparc-t1/index.html)