The Design of the Microarchitecture of UltraSPARC™-I

MARC TREMBLAY, MEMBER, IEEE, DALE GREENLEY, AND KEVIN NORMOYLE

Invited Paper

The realization of a high performance modern microprocessor involves hundreds of person-years of conception, logic design, circuit design, layout drawing, etc. In order to leverage effectively the 5–10 millions of transistors available, careful microarchitecture tradeoff analysis must be performed. This paper describes not only the microarchitecture of UltraSPARC-I, a 167 MHz 64-b four-way superscalar processor, but more importantly it presents the analysis and tradeoffs that were made "en route" to the final chip. Among several issues, the in-order execution model is compared with alternatives, variations of the issue-width of the machine as well as the number of functional units are described, subtle features that are part of the memory hierarchy are explained, and the advantages of the packet-switched interconnect are exposed.

I. INTRODUCTION

A. Overview of UltraSPARC-I

UltraSPARC-I is a highly integrated 64-b, four-way superscalar processor targeted at running real life applications 2.5–5× faster than the previous SPARC processors. To achieve this goal, several processor architectures, as well as a plethora of microarchitecture features, were investigated. This paper describes *why* we settled on the current architecture implementation.

In order to quantify *why* we should use a certain execution model, or a certain branch prediction scheme, or a certain mix of functional units, etc., we wrote an accurate performance simulator with tunable parameters so that many variations could be simulated [1]. A novel sampling methodology [2] was used to speed up simulations so that turnaround time for simulating a set of about 30 applications (including SPEC92) on a new machine model would take only a few hours. No less important, the impact of the features on cycle time was evaluated through circuit simulation (Spice) of the main paths affected.

The block diagram in Fig. 1, shows a high level representation of the microarchitecture of UltraSPARC-I. The front-end of the machine responsible for prefetching, decoding,

Manuscript received August 2, 1995; revised August 24, 1995.

The authors are with SPARC Technology Business, Sun Microsystems Inc, Mountain View, CA 94043 USA.

IEEE Log Number 9415192.

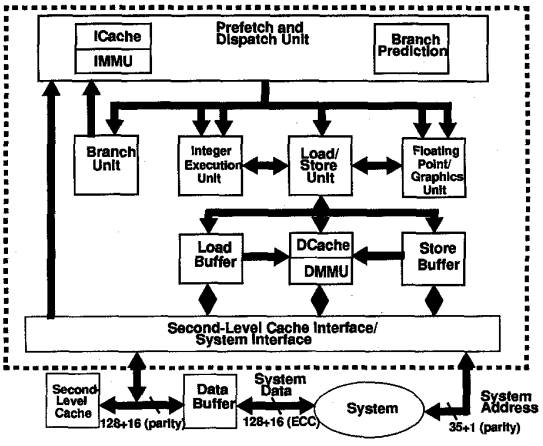


Fig. 1. UltraSPARC-I block diagram.

and dispatching instructions (stages F, D, and G in Fig. 2) as well as predicting branches and their target, is capable of sending four instructions per cycle to the nine functional units even in the presence of conditional branches. Section IV describes how we achieved this and describes some of the alternatives that were considered. The core of the machine consists of nine functional units. In Fig. 1 we show how many instructions can be dispatched to these units every cycle. For instance, two instructions (two arrows) can be sent to the floating-point and graphics block (which contains five distinct functional units). All functional units are fully pipelined except for the floating-point divide and square root unit. Instructions are dispatched in order but long latency instructions (e.g., FP divide/square root, load misses, etc.) are allowed to complete out-of-order with respect to other classes of instructions, so that their latency can be hidden.

The backend of the processor is composed of the load buffer, the store buffer, the data cache (also referred to as the first level cache), the Data Memory Management Unit (DMMU), and the second-level cache (also referred to as external cache) controller. These units combine to provide

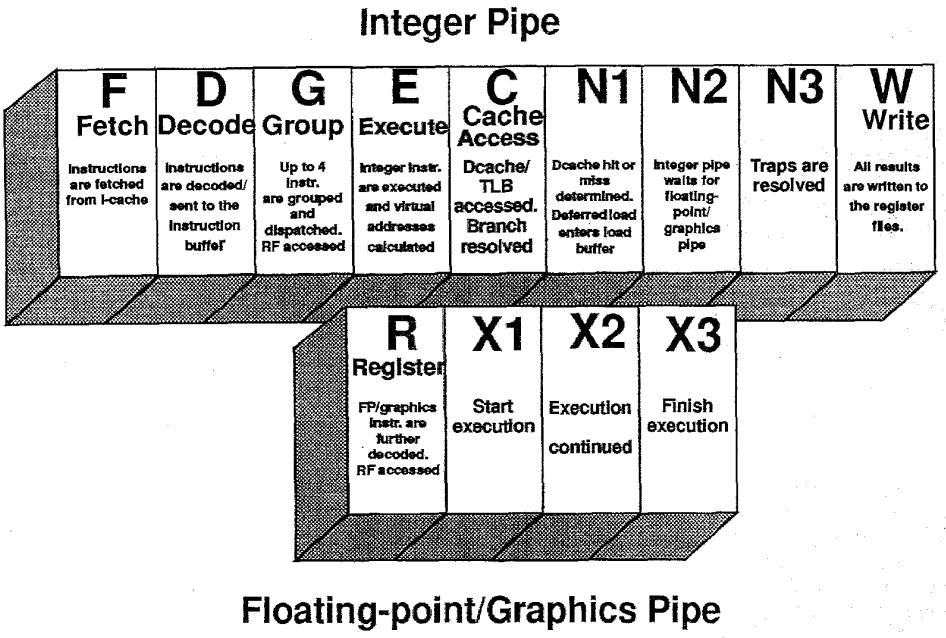


Fig. 2. UltraSPARC-I pipeline.

the necessary data bandwidth to the functional units. A nonblocking memory system in conjunction with scoreboarded loads allow UltraSPARC-I to *sustain* a bandwidth of one access per cycle to the large (up to 4 Mbytes) external cache.

The UltraSPARC Port Architecture (UPA) is a high bandwidth packet switch interconnect supporting several coherent masters. The 16-byte wide (128 b) data bus is protected using 16 b of ECC and is decoupled from the address bus. Through extensive microarchitecture support, UltraSPARC-I can sustain a bandwidth of 600 Mbytes/s from main memory back to main memory (or to the frame buffer).

The execution model we chose for UltraSPARC-I is compared against out-of-order execution models in Section II. The impact on performance from the issue width and the mix of functional units is covered in Section III. Tradeoffs in the design of the front end (prefetch and dispatch unit) as well as in the back end (caches, load buffer, store buffer, etc.) are explained in Sections IV and V. System interconnect design decisions are described in Section VI. Finally, Section VII describes the reasoning behind implementing multimedia functions (the visual instruction set (VIS)) on-chip.

II. Execution Model

The execution model of a processor has a major impact on the whole design process. Not only does the execution model define the backbone of the pipeline, but it also affects the cycle time of the machine, the complexity of validating the logic, the time to tapeout, the die size, etc. In order to study which execution model would be best for UltraSPARC-I, we investigated a variety of proposals, each with a certain degree of "out-of-orderness." For instance, given a not-to-exceed die size in a 0.5 μm technology (around 315 sq mm), an aggressive out-of-order execution processor with a unified 64-deep window was thoroughly simulated. Similarly, several instances of a fast superscalar processor with a simple in-order execution model were simulated.

In general our simulations showed that an out-of-order superscalar processor achieved an instructions per cycle (IPC) around 30% higher than a strict in-order machine of comparable width, when simulating "old" integer code. In recent years, advances in optimizing compilers, in particular global code motion, have allowed in-order machines to close the gap between what an out-of-order machine can achieve (in terms of IPC). Compilers have the luxury of being able to look at a window of instructions larger than what can be achieved in today's hardware technology (up to 64 instructions [4]), which means that more aggressive code motion can be done. Trace scheduling [6], [7] and superblocks [8] are two examples of global code motion techniques. Simple architecture additions, such as speculative loads [22] and a nonblocking memory subsystem were added to UltraSPARC-I in order to make it worthwhile for the compiler to hoist instructions so that part of their latency, or even their full latency, can be hidden. The combination of profile feedback optimizations and novel static branch prediction heuristics [9], achieving in some cases accuracy greater than 80%, mean that an optimizing compiler can perform code motion on the most likely instruction path. For nondeterministic events, such as cache misses, the compiler can generate code assuming that the latency of an access will not hit the first level cache (but will hit the second level). Through microarchitecture support for a nonblocking memory subsystem (e.g., scoreboarded

loads) the hardware can deliver data at the same rate as if the data had been in the first level cache and can do so while completely hiding the load latency. Transforming nondeterministic events into deterministic ones is also possible. For instance, software prefetch can be used so that data is most likely in the first level cache.

Given that current generation machines are significantly different from the previous generation, recompilation seems to be necessary for maximum performance even for an out-of-order machine. SPEC92 numbers published by vendors have increased significantly with newer binaries [4]. Simulation of recompiled code and simulations of rescheduled code [14] (more in Section III), indicate that the IPC difference between an in-order machine and an out-of-order one on SPECint92 would only be 15% if aggressive cross block scheduling is used. For scientific computing, where instruction level parallelism (ILP) is easier to exploit, we did not notice any significant advantage. Well known compilation techniques, such as software pipelining [11] can generate optimized code which is not improved through hardware reordering. Similarly, applications which rely heavily on indirect data references (e.g., database applications) did not see a gain in IPC from out-of-order execution.

The impact of the more complex out-of-order logic on cycle time was also evaluated. For instance register renaming, typically accomplished through an associative lookup or through a mapping structure, affects the register file access time if tied to the same pipeline stage [4], or alternatively, an extra stage in the pipeline may be required [5], resulting in a loss of 2–4% on SPECint92 depending on the branch prediction scheme used. The instruction selection logic in a unified window scheme is complicated by the fact that the processor must recover quickly when a misprediction or a trap occurs. Selecting one of 64 instruction, or two out of two banks of 32 instructions, is an operation that exists only in out-of-order processor. Large fanouts on result buses and complex logic for picking the "right" instruction (selecting instructions influencing the height of the dependency graph) was evaluated to be a cycle time limiter. Finally the unit responsible for retiring instructions in order so that precise exceptions be supported also impacts cycle time since it typically requires more write ports into the register file than an in-order machine, so that a burst of instructions can be retired quickly. Otherwise, an unbalanced machine creates excessive resource overflows (e.g., queues full, out-of-ports, etc.), delivering less performance. Complex FIFO structures required for the retirement logic have been described in various papers [12], [13]. A simplified in-order execution machine renders these structures unnecessary.

Considering the critical paths above and accounting for the fact that more logic is required, resulting in a larger die, we estimated a global impact on cycle time of around 20%. For a 0.5 μm technology this represents the difference between 167 MHz (our goal for UltraSPARC-I) and 133 MHz. For a 0.35 μm technology, the difference is 50 MHz (250 MHz versus 200 MHz).

Finally, another key criteria that must be considered when comparing two architectures is the impact on schedule. The performance of microprocessors since the early days has steadily increased at a rate of 1.5–1.6 per year (around 2 $\times$ every 18 months), or around 4% per month. Because of the additional complexity, larger die, greater pressure on cycle time, etc., we evaluated that implementing an out-of-order machine would cost us between 3–6 months. Additionally, bring up time, due to more complex testing and functional verification, would be lengthened. Bringing a processor to market 3–6 months later represents a performance loss equivalent to 12–26%.

UltraSPARC-I dispatches and executes instructions in the same order as they appear in the code. Every cycle, the grouping logic dynamically computes how many of the top four instructions sitting in the instruction buffer can be dispatched to the functional units. Instructions are allowed to complete out-of-order so that long latency operations can be bypassed by shorter ones. For instance, loads to the data cache, to the external second level cache, or to main memory are allowed to finish out-of-order with respect to other classes of instructions such as integer operations, floating-point instructions, etc. Similarly, floating-point divides and squareroots can finish out-of-order with respect to all other instructions.

The completion unit allows instructions of different latencies to update the register file in an orderly manner, thus presenting a precise state to the operating system when an exception or interrupt occurs. Exceptions related to loads, such as TLB misses, unaligned accesses, protection violation, are all detected at the beginning of the pipeline. The only error not accompanied by precise state is due to a parity error, but in this case recovery is not possible since the operating system terminates the process anyway.

The UltraSPARC-I execution model, while maintaining the simplicity of an in-order execution machine, takes advantage of a nonblocking memory system and scoreboarded loads in order to hide long latencies operations. The additional gain in IPC ($\sim$15%) obtained from a more complex out-of-order execution model is more than offset from the benefit in clock rate ($\sim$20%) and schedule ($\sim$12–26%).

III. Issue-Width and Functional Units Mix

UltraSPARC-I is a four-way superscalar unit. Every cycle, groups of 0, 1, 2, 3, or 4 instructions can be issued to the nine functional units residing on the chip. The issue-width and the functional unit mix have a large influence on the CPI and cycle time of a machine, therefore their selection must be backed by extensive simulations. Simulations of possible configurations for UltraSPARC-I ranged from a uniscalar machine to a 5-scalar machine. For each configuration, various mixes of functional units were simulated and the full SPEC92 benchmark suite as well as large applications such as Hspice, database traces, verilog, etc., were run to quantify the impact on performance. This methodology is very similar to the one recently described in [15], except that we also investigated the impact of

Table 1 Improvement Over an Uniscalar Machine

Issue-width	Code compiled for uniscalar	Rescheduled code for target processor
uniscalar (normalized)	1X	1X
2-scalar	1.14X	1.46X
3-scalar	1.26X	1.78X
4-scalar	1.33X	1.83X

Table 2 Speedup Obtained by Adding an ALU

Issue-width	Code compiled for uniscalar	Rescheduled code for target machine
2-scalar	1.06X	1.12X
3-scalar	1.13X	1.25X

target specific code (while an out-of-order machine was considered in [15]). As shown in Table 1, under the column labeled "Code compiled for uniscalar," the improvement on existing code over a uniscalar machine for a 2, 3, and 4-scalar machine is 14%, 26%, and 33%, respectively, over a uniscalar processor for SPECint92.

An important factor to consider when measuring such variations is which compiler is used to generate the binaries feeding the simulation. The numbers shown in the first column in Table 1 were obtained from binaries generated for a uniscalar processor. A compiler which has knowledge of the underlying machine can generate code tailored to the width of the machine. Optimizations such as reorganization of the order in which the instructions appear, predicated execution, as well as cross block scheduling translate into significant gains for superscalar processors [10]. In order to measure how much performance could be gained for each configuration with optimized code, we developed a "rescheduler." The rescheduler has the capability to look at a large window of instructions (128, 256, 512, etc.) generated from a trace and can regenerate an optimized trace by moving instructions by tens of positions while still respecting data dependencies. Several parameters can be set to bound the code motion. For example, the number of branches allowed to be passed can be limited. In this way, aggressive compiler optimizations can be simulated in a much easier manner than by modifying the compiler.

Using the rescheduler, we tailored binaries for each configuration in Table 1 and obtained a much better improvement as shown in the same figure under the label "rescheduled code." The speed-ups reached are 46%, 78%, and 83% for a 2, 3, and 4-scalar machine, respectively. Based on such simulations and based on analysis of the impact of the issue-width on critical paths, we set the issue width of UltraSPARC-I at four.

Similar experiments were conducted to determine the functional unit mix. An example is given in Table 2. The improvement obtained from adding a second integer ALU for a 2-scalar and 3-scalar machine is shown for both existing and rescheduled code.

The improvement on the code scheduled for a uniscalar processor is 6% for the 2-scalar machine and 13% for the 3-scalar machine. For rescheduled code, the gain is doubled to 12% and 25%, a much more attractive benefit. This methodology was used to arrive at the current mix of functional units shown in Table 3.

Table 3 Functional Units on UltraSPARC-I

Functional Units	
64-b integer ALU	2
Load/Store	1
Branch	1
Floating-point adder	1
Floating-point multiplier	1
Floating-point divider/square root	1
Graphics adder	1
Graphics multiplier	1

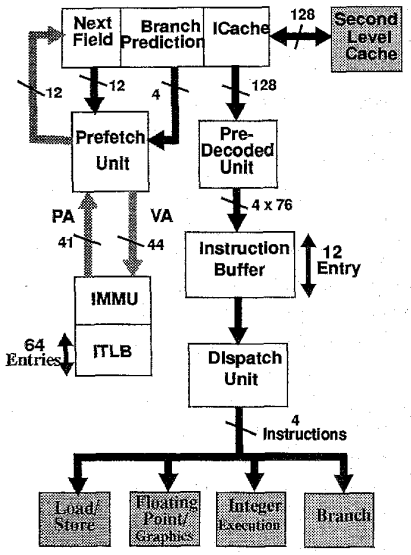


Fig. 3. UltraSPARC-I front-end.

IV. Front End

The front end consists mainly of the instruction cache, the branch and target prediction mechanism, the instruction translation lookaside buffer (ITLB), the instruction buffer and the dispatch unit (Fig. 3).

A. Instruction Cache

The instruction cache (I-cache) is somewhat unique in its organization. It is described as 16 kB in size, if one counts a SPARC instruction as 4 bytes. However, stored with each instruction in the I-cache are predecoded bits that are used for instruction fetching. The I-cache has qualities of both a direct-mapped and a two-way set-associative cache. It can be considered two-way in that a particular address can be present in two different locations in the cache. It is considered direct-mapped in terms of access time in that the "set" to access is predicted ahead of time, so there is no comparison function in the access path. This is a cycle-time and power dissipation advantage in that only one set is accessed. The "set" prediction bit is obtained from the Next

Field RAM, explained in Section IV-B. In the rare case of a set mispredictions, a two-cycle fetch penalty occurs, which does not necessarily translate into a loss of performance due to the instruction buffer. This approach also allows the I-cache to be both physically indexed and physically tagged, simplifying coherence issues. The I-cache line size is 32B (eight instructions); there is no subblocking. An I-cache fill takes two clocks since the interface to the rest of the memory system is 16B wide.

B. Branch Direction and Branch Target Prediction

Every cycle up to four instructions can be prefetched from the instruction cache and sent to the instruction buffer. Each line in the I-cache contains eight instructions (32 bytes). Every pair of instructions has a 2-b branch prediction field (Fig. 4) which maintains history of a possible branch in the pair. The four prediction states are the conventional strongly taken, likely taken, strongly not taken, and likely not taken [16]. The advantage of the in-cache prediction scheme is that it avoids the alias problems encountered in branch history buffer and other similar structures [17]. Implemented in this way, every single branch in the I-cache has its dedicated prediction bits (ignoring the rare case of branch couples), which translates into a high successful prediction rate of 88% for integer code, 94% for floating-point (SPEC92), and 90% for typical database applications.

Every group of four instructions in the cache also has a "next field" (Fig. 4) which is simply a pointer to where the prefetcher should access instructions for the very next cycle. In the case of sequential code or for code with a branch predicted not taken, the next field points to the next four instructions in the cache. The next field will contain the I-cache index (including the set) of the branch target if a branch is predicted taken. The advantage of this scheme is that the next field can always be fed back to the I-cache without qualifying a possible branch thus saving levels of logic. The next field mechanism is capable of handling a branch every cycle even if previous branches haven't been resolved. Due to the four cycle branch resolution latency, UltraSPARC-I can speculate five branches deep (including the branch being resolved) representing up to 18 instructions.

V. Memory Subsystem

The memory hierarchy consists of the instruction cache (I-cache), data cache (D-cache), and external cache (E-cache). The load buffer and store buffer provide an interface between the Integer and Floating-point functional units and the memory system for data references. There is both an instruction and data memory management unit (IMMU and DMMU) which provide dedicated virtual-to-physical address translation for instruction and data references, respectively.

A. MMU's

UltraSPARC-I supports a 44-b subset of the full 64-b virtual address space. This reduction was done due to both die size limitations and timing impacts. UltraSPARC-I would have been approximately 3–5% larger to support a 64-b virtual address as this affects not only the MMU's, but also affects instruction fetching, branch resolution, and trap recording datapaths. Additionally, one of the top critical paths in the machine involves the generation of the virtual address in the E-stage of the pipeline (Fig. 2) through a fast, dynamic adder and distribution to other parts of the chip (the branch unit for register-based control transfer instructions (CTI's), the D-cache and D-cache tag RAM's for data references, the Data TLB for virtual-to-physical address translations). So, even though all 64 b are generated in order to check that the address is not out-of-range, there is no need to distribute the upper 20 b, nor optimize their timing. From the OS point-of-view, 44 b of virtual space are sufficient for the lifetime of the processor.

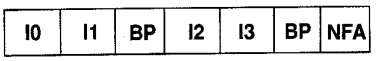


Fig. 4. Logical line in the instruction cache.

Each MMU has a 64-entry, fully associative TLB which can perform one address translation per cycle. Four page sizes are supported: 8 kB, 64 kB, 512 kB, and 4 MB. Larger page sizes are useful in mapping large contiguous regions of memory like I/O space, frame buffers, and parts of the kernel. Without them, we are only able to map 1 MB (128 * 8 kB) of physical memory, which would otherwise degrade performance of larger External caches (e.g., 75% of a 4 MB E-cache would be accessible at any given time only after taking a TLB miss). We considered set-associative approaches for the TLB's, but found it difficult to conveniently support multiple page sizes without having dedicated TLB's for the larger pages. The Instruction and Data TLB's are identical mainly to minimize the design resources required, and the number of entries was chosen based on performance analysis on a wide range of benchmarks.

TLB misses are mainly handled in SW as fast traps, with a fair bit of HW support provided in the MMU's. This decision was made for two main reasons: 1) offer the flexibility of a software solution to the operating system so that various paging mechanisms can be supported, and 2) prior experience with hardware TLB miss processing showed it to be complex, prone to bugs, and difficult to verify.

B. Data Cache

The D-cache is a 16 kB direct-mapped cache. It has a 32B line size, with 16B subblocks. It is virtually indexed and physically tagged. It operates on a write-through, no write-allocate policy. It is nonblocking so that D-cache misses and other conditions which delay memory operations (specifically loads) do not necessarily penalize subsequent instructions.

We chose a direct-mapped design for simplicity, fastest access time, and low latency, which results in a 1-cycle load-use penalty. These advantages, combined with an ag-

gressive external cache implementation, offset the slightly higher performance that can be obtained though a set-associative cache. Once we decided to implement a direct-mapped design, our choice of cache sizes was limited to powers of 2 (e.g., 8 kB, 16 kB, 32 kB, etc.) and here die size limitations were decisive. Due to the 8 kB page size, the D-cache must be virtually indexed so as not to wait for the TLB to produce the extra physical address bit. We depend on the OS to not create virtual aliases through its page mapping scheme, so the only complexity comes from maintaining cache coherence with the E-cache and other processors. In such cases, a purely physical address is provided and used to interrogate both possible "virtual" locations. This requires slightly more bandwidth, but since the D-cache obeys the principle of inclusion with respect to the E-cache, this is an acceptable tradeoff.

Performance simulations indicated that a 16B line size was optimal, but area constraints forced us to implement a 32B line size, with 16B subblocks, basically halving the size of the D-cache tag RAM. Given that the interface to the D-cache is 16B wide, we are able to do a subblock fill every cycle.

We chose a write-through, no write-allocate policy over a strict write-back or variations which combined write-through/write-back policies via modes or on a page-by-page basis. We judged the write-through design to be of much less complexity and pose fewer risks for a bug-free design. There are three main reasons why a write-through design was acceptable on UltraSPARC-I:

- Lines in the D-cache are always clean, which means that when loads are scheduled for the latency of the E-cache and bring data in the D-cache every cycle, dirty victims do not have to be flushed back to the next level. Sustaining one load per cycle to the external cache was a key design requirement for UltraSPARC-I in order to run large applications fast.
- An E-cache must be present (i.e., there is no option to run UltraSPARC-I with primary caches only). This means that Stores need not be reflected on the memory interconnect and that memory does not have to do read-modify-write operations.
- The E-cache is pipelined so that one operation (either load, store, or instruction fetch) can be completed every cycle. This, combined with techniques to be discussed later, provides sufficient store bandwidth to reflect all stores to the E-cache.

The D-cache tag RAM is dual ported, unlike the data RAM, which is single-ported. It has two independent ports: a read port and a read/write port. Providing two ports allows a tag look-up to occur for a younger load or store in parallel with a D-cache fill for an outstanding miss or an invalidation (e.g., due to an external snoop). This basically allows the pipeline to process 1 load per cycle from either the D-cache or the E-cache.

C. Load Buffer

The Load Buffer is a nine-entry FIFO-style queue that enables the machine to support nonblocking loads. We chose to implement such an execution model in an effort to hide as much of the memory latency as possible. This scheme is less attractive in single-scalar designs, but in UltraSPARC-I every clock cycle the machine is waiting for a load to complete means that up to four instructions could not start execution. Thus loads are completed out-of-order with respect to subsequent instructions which do not depend on the load result. We chose not to allow load operations to return out of order with respect to other load, mainly due to complexity and cycle time reasons.

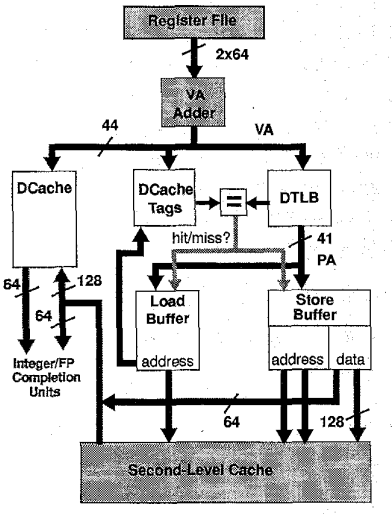


Fig. 5. Interaction of the load buffer and store buffer with the D-cache and E-cache.

If a Load cannot be returned immediately from the D-cache, it is placed on the load buffer (Fig. 5). This typically for D-cache misses, but can also occur for read-after-write hazards, noncacheable references, nonperformance critical instructions, and other reasons. Nine entries were implemented so that the machine could maintain a throughput of one load per cycle, even in the presence of D-cache misses (assuming an E-cache hit). This is an attractive feature, especially for most Floating-point loops and other latency tolerant algorithms, as the compiler can schedule instructions for the longer E-cache latency, as opposed to the much shorter D-cache latency. This effectively allows such programs to perform as if they were running out of a very large primary cache. We made no effort to cover the latency to main memory since such accesses are usually infrequent due to the effect of the much larger external cache. Additionally, scheduling code to do useful work in the 20–40 cycles behind an E-cache miss is quite difficult (the E-cache is nonblocking but only one E-cache read miss can be outstanding)

D. Store Buffer

The Store Buffer is an eight-entry FIFO-style queue that provides a temporary holding place for store operations until they are "committed" and can update the D-cache and/or E-cache. Each entry holds a 64-b datum, as well as

its corresponding physical address and control information. The Store Buffer was required to have enough storage for 64 b of data, in order to support the Visual Instruction Set's Block Store operations. Thus eight entries was the minimum. Simulations indicate that additional entries did not give a noticeable performance boost, given some of the features described below.

Given that the D-cache is write-through, all cacheable stores must be reflected to the E-cache. Unlike loads, stores are done in two steps: first, the E-cache is checked for hit/miss; second, the E-cache is updated with the appropriate data. The E-cache tag and data RAM's are decoupled (i.e., they are tied to separate address and data wires) so that a tag check can occur in parallel with the E-cache data update of an older store (Fig. 5). Since the E-cache tag check can be started as soon as the physical address is available, we are typically able to maintain a throughput of one store per clock.

There are several additional techniques that we use to minimize both E-cache tag and data RAM bandwidth requirements. The tag check and data write for a particular store are not explicitly synchronized, which allows for some variation in access patterns. To support this, the E-cache hit/miss indication is queued in the Store Buffer for eventual use. In the rare case that the state of the E-cache line is changed before the store can be completed, the tags are simply reaccessed. Additionally, consecutive stores to the same E-cache line (64 b) typically require only a single tag check, further reducing needed tag check bandwidth. Lastly, a compression feature is implemented which combines the last two Store Buffer entries into a single operation if they target the same 16 b subblock. There are no restrictions as to memory alignment nor on how many entries can be combined into a single operation. Thus data write transactions can be minimized significantly. This feature came mostly for free since it was already necessary to be able to expand 64-b store operations to the 128-b width of the interface to the E-cache.

D. External Cache

The E-cache lies between the primary caches (I-cache and D-cache) and main memory. It is direct-mapped and both physically indexed and physically tagged. E-cache sizes between 512 Kb and 4 Mb are supported. It operates on the expected write-back, write-allocate policy. The 64-b line size (no subblocking) and modified, own, exclusive, shared, invalid (MOESI) coherency algorithm were chosen based on system considerations which are discussed later. All E-cache control is handled on chip, while the tag and data arrays are implemented in fast, synchronous SRAM's.

The I-cache and D-cache are always kept consistent with the E-cache, thus obeying the principle of inclusion. Inclusion, as well as the fact that the D-cache operates on a write-through policy, means that the E-cache can act as a filter for cache coherent requests from other processors or I/O devices, thus reducing the load on the primary cache and eliminating the need for duplicate tag arrays. Since the D-cache cannot hold "dirty" data, the latency to access data in another processor's cache is minimized.

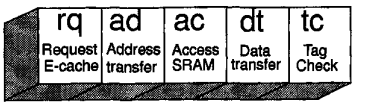


Fig. 6. External cache pipeline.

A virtually indexed E-cache was considered briefly, as this hypothetically allows an access to begin a cycle earlier in the pipeline (i.e., before the TLB translates from virtual to physical address). This was rejected for a variety of reasons. For one, E-cache accesses would have to speculate that a D-cache miss would occur in order to gain an extra cycle; too much speculation produces extra bandwidth on the E-cache bus preventing other nonspeculative requests from being executed, too little speculation reduces the benefit. Second, it requires the OS to manage virtual page aliases on E-cache size boundaries. Third, it complicates the cache coherency problem by requiring duplicate and/or extra virtual address bits to be passed around on the interconnect.

We chose to implement a direct-mapped design over 2, 3, or 4-way set-associative approaches. As mentioned earlier with respect to the D-cache, there is a tradeoff of complexity and latency versus hit rate, although in this case response to cache coherency traffic is also a consideration. It is well known that the performance of direct-mapped caches is more susceptible to the allocation of memory space and virtual-to-physical address mapping [18]. On the other hand, the complexity (added pins) of fetching multiple tags from external SRAM's or the added circuitry needed to predict the set [4] was not attractive especially since in the latter case, the penalty for mispredicting the way can offset the performance advantage of an associative cache (seven cycles for the MIPS R10000).

The additional latency for an access that misses in an internal cache (I-cache or D-cache) and hits in the E-cache is six cycles. For a 166 MHz processor, 6 ns, synchronous SRAM's are used, allowing a throughput of 1 read or write per cycle, which is crucial to supply the bandwidth required by the internal caches and cache coherency traffic in SMP configurations. Both tag and data arrays use an identical SRAM, which simplifies timing analysis and reduces cost. There is a penalty when switching from reads to writes as a dead cycle is inserted on the bidirectional E-cache bus to avoid electrical issues with overlapping drivers. The SRAM includes a one-entry delayed write buffer which reduces this penalty by one clock cycle. Additional entries in this write buffer could have reduced the penalty to zero cycles, but SRAM vendors were unwilling to modify "commodity" type parts within an acceptable cost structure. The E-cache pipeline is shown in Fig. 6. One should note that the AD, AC, and DT stages occur off-chip, while the RQ and TC stages occur on-chip.

Both data and tag arrays are protected with byte parity. Due to the high reliability of SRAM's, ECC protection

was judged unnecessary. Reasonable ECC (e.g., for a 64-b word) requires a read-modify-write for any write smaller than a word, a major performance bottleneck (e.g., for writing pixel data).

VI. The UltraSPARC Port Architecture (UPA)

A. Pin Allocations

Chip level interfaces are driven by hard physical constraints. Given our die size goal, and use of standard I/O pads and BGA packaging technology, a total of around 320 I/O signals were deemed available (UltraSPARC-I ended up having 333 signal pins and 187 power and ground pins).

A 64-b E-cache interface with a 64-b shared address/data system interface was originally considered. Simulations indicated that large applications, graphics routines, as well as functions used in network protocols (e.g., check summing large amount of data) would quickly saturate such a bus. These performance effects, although not fairly represented by SPEC92 benchmarks, are critical for real-world applications. The goal for the system interface was simple: increase performance for a wide range of user applications by minimizing latency and maximizing throughput for both memory and noncacheable transactions [19]. No easy goal when one usually comes at the expense of the other.

A wide 144-b E-cache data interface for low latency L1 cache fill was chosen. That needed to be coupled with a wide path to memory to keep E-cache fill and evict time low. Keeping the E-cache bus busy twice as long for these events (with a narrower bus) would eventually stall the processor otherwise, since L1 and E-cache misses tend to occur in bursts.

An E-cache built out of SRAM's with two bus ports (one port for the processor, one port for the system) was rejected due to its high cost and lack of availability. The other alternative, adding another wide 144-b bus on the processor was also rejected due to lack of signal pins available. The solution was to connect the external cache to memory without going through the processor, by adding two cheap external buffer chips (Fig. 7 and Section VI-B). It was critical that this addition did not impact our ability to run the fully pipelined E-cache at the processor clock rate. This separate path for data resulted in a relatively small number of pins on the processor for the system interface. A separate system address bus allows us to send address and data in parallel to the system, which is especially important for sustaining high throughput to graphics subsystems.

Thirty-seven pins are used for the system address, ten for handshaking on request/reply handshaking, five for arbitration, five for controlling the buffers, and four for ECC reporting. The first cycle of the address packet sends all bits necessary for initiating the RAS cycle at the DRAM. The address packet includes a 16-b byte enable field, which is useful for efficiently supporting random pixel writes to a graphics frame buffer. A distributed arbitration protocol is used on the address bus, saving cycles on every transaction, compared to an external arbiter. Even though the system bus is packet switched, there is no arbitration for, or use of, the address bus when returning a data packet. Separate radial wires provide the necessary handshake for completing data transfers.

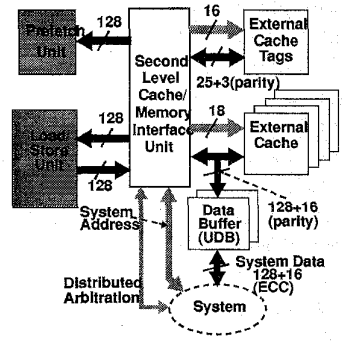


Fig. 7. System interface for UltraSPARC–I.

The separate address bus keeps data bus utilization lower, which helps minimize the average latency for E-cache fills for both uni- and MP systems. Bus contention increases latency, which has a cumulative negative throughput effect. Longer latency usually results in increased distance between requests, due to program data dependencies. The difference between "peak" and "realized" bandwidth for many systems has traditionally been caused by long latencies for individual events.

B. UltraSPARC Data Buffer (UDB)

Usage of external buffers allowed us to add FIFO's and ECC check/generate logic without area or latency penalty. The UDB's are pad limited, making "extra" active area available, and data transfers take one system clock to pass through. So ECC can be computed here, for "free." For cost reasons, the UltraSPARC Data Buffer consists of two ASIC's. Since the rest of the system was built from similar ASIC's, the UDB does not need the additional performance of a full semicustom design.

We looked at running the UDB with two clocks, but decided to only use the system clock for simplicity. Movement of store data into the UDB is under processor control, at which point unloading it is the responsibility of the system. Queues are used to hide the latency inherent in backflow control signals from the subsystem receiving the data. This allows UltraSPARC-I to sustain the peak data bus bandwidth for noncacheable stores to frame buffer, for example. In addition, buffers are available for E-cache fill, E-cache writeback, 64-byte block stores, and E-cache copybacks on behalf of other cache-coherent requestors.

C. Flexible Interconnect for Low Cost Uniprocessors, High Performance Multiprocessor Systems

In order to keep snoop latency low and predictable for multiprocessor systems, a duplicated set of the external cache tags can be tightly coupled to the system controller so that only snoop hits are sent to the individual processors. MP systems without dual tags like these can see their performance decrease as traffic increases. The average

latency for snoops can increase, because of collisions with processor access to the E-cache tags. which causes the average memory read latency to increase. Also, some systems end up being throttled by limited throughput when snooping the E-cache tags. Having dual tags eliminates these problems, providing consistent high throughput and low latency.

Uniprocessors can work without dual tags due to the low snooping traffic. Only cache coherent IO activity requires snooping the E-cache tags. UltraSPARC-I was designed to support systems with and without dual tags.

UltraSPARC-I targets a wide range of designs, so a flexible system interface was important. As processors integrate more and more system functions, it is easy for the unwary to hardwire some interface behavior that makes implementations of SMP and MPP systems difficult. For instance, UltraSPARC-I doesn't snoop addresses on the shared bus. This would force an ordering of coherency events dependent on their arrival on that single shared address bus. The processor only receives snoop requests after the system decides the processor cache needs to be invalidated, or needs to provide the data. Dual tags or directories can be used to make this decision.

Uniprocessor systems with cache coherent IO DMA can interrogate the processor cache for every DMA to memory though. The overall IO DMA bandwidths don't cause much E-cache bandwidth to be lost, and the processor responds fast enough to not impact the deliverable IO DMA bandwidth.

D. Many Outstanding System Events

The current implementation of the UPA supports eight outstanding noncacheable stores, enough to offset the latency for returning a handshake signal indicating an entry has been unloaded from the input queue in the system, without creating a bubble in peak data delivery.

There can also be two outstanding block stores, one outstanding writeback, and one outstanding E-cache miss. The external cache control unit keeps track of three E-cache misses internally, so there is a fair amount of overlap in E-cache miss processing. Only the processor pin-to-pin delay is serialized for an E-cache miss from a single processor.

Given the latency inherent in a typical memory system, UltraSPARC–I can demand over 350 Mbytes of memory bandwidth. Adding 175 Mbytes of writeback bandwidth creates the need for a wide banked memory. The 16-byte wide UPA data bus running at 83 MHz can accommodate several processors by delivering 1.3 Gbytes of data to clients.

VII. Multimedia Support

Graphics speed has a big effect on a workstation user's perception of performance. Graphics functionality is growing increasingly more sophisticated including desktop video for teleconferencing and broadcast quality viewing, 3D visualization and animation, image manipulation for desktop publishing, etc.

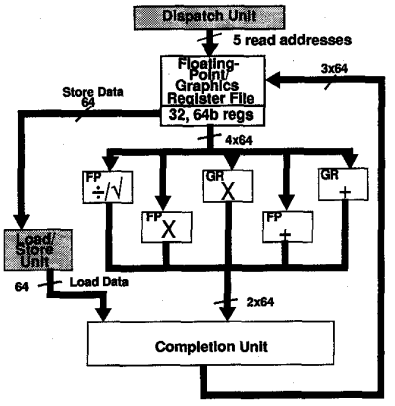


Fig. 8. Floating-point and graphics unit.

Up until now, specialized graphics hardware was required for these applications. Typically, additional functionality could be provided to the base machine by adding one or more graphics cards. For example, MPEG-1 decompression could be achieved by adding one card, while 3D visualization could be supported by adding another card. Implementing support for these applications directly on the processor may remove the need for additional graphics cards, leading to better overall system cost and freeing precious I/O slots.

The lack of a standard platform that supports these features has hindered the development of multimedia application software. With UltraSPARC-I, we saw an opportunity to provide a standard multimedia capability for future SPARC systems with only a 3% increase in the die area. The 30 new instructions from the visual instruction set (VIS) can be used as other RISC instructions on UltraSPARC-I. There is no need to perform memory mapped I/O or to access I/O devices.

Implementing VIS on the processor also means that the performance scales with frequency upgrades. Typically, processor frequency follows an aggressive curve due to gate shrinks and/or full process shrinks. These are typically not available or do not improve as rapidly on ASIC's. Scaling also appears in multiprocessor systems. Many multimedia applications lend themselves well to multithreading, which can attain, in the ideal case, linear speedup with N processors.

Multimedia instructions are executed by two specialized execution units in the floating point datapath (See Fig. 8). These RISC style instructions provide the core operations needed by multimedia algorithms. Specific algorithms such as MPEG are implemented by software libraries using these instructions. The execution units are fully pipelined so that each cycle 2 VIS instructions can be issued.

The execution units were added to the floating-point unit mainly for four reasons. First, more registers are available because graphics data can be stored in all 32 FP registers while addresses and loop counts are stored in the integer registers. Second, floating-point units are typically not used concurrently with VIS instructions, which

means that the issue slots that would normally be used for floating-point instructions can be used for VIS instructions allowing the maximum parallelism to be achieved (four instructions per cycle). Third, some instructions have a latency of three clocks which fits naturally into the floating-point pipeline design. Fourth, the basic cycle time of the machine is based around key datapaths components dictated by the integer side of the processor (e.g., ALU's, data cache access, etc.). Implementing VIS instructions on the integer side would have introduced extra gate levels in the adder (allowing intermediate carries to propagate for normal adds), it would have added new functional units (e.g., four signed multipliers), and more bypasses into critical muxes would have been needed.

A complete description of VIS is given in [20] while its usage for a broadcast quality MPEG player is described in [21].

VIII. Conclusion

The realization of UltraSPARC-I required 5.4 million transistors for the circuits, 600 computers (1000 processors) for simulation, 2 Tbytes of disk space, 300 person-years, etc. While these "resources" contribute to making the implementation of a processor successful, the initial phase of the project, more precisely the architecture and microarchitecture definition, have a large impact on not only how the processor will perform but also on how long the implementation will take and how expensive the die is going to be. In this paper we have presented how the microarchitecture of UltraSPARC-I was derived by presenting various proposals and alternatives that were investigated before settling on the final design.

Acknowledgment

The authors would like to acknowledge Les Kohn and Bill Joy for their contribution to the microarchitecture, and the whole UltraSPARC team for actually delivering the chip! The authors would also like to thank Don Van Dyke and Robert Garner for constructive comments on the paper.

References

[1] M. Tremblay, G. Maturana, A. Inoue, and L. Kohn, "A fast and flexible performance simulator for micro-architecture trade-off analysis on UltraSPARC™-I," in *Proc. 32nd Design Autom. Conf.*, San Francisco, June 1995, pp. 2–6.

[2] G. Lauterbach, "Accelerating architectural simulation by parallel execution of trace samples," Sun Microsystems Labs., Tech. Rep. SMLI TR-93-22, Dec. 1993.

[3] D. Greenley *et al.*, "UltraSPARC: The next generation superscalar 64 bit SPARC," in *40th Annu. Compcon,* Mar. 1995, pp. 442–451.

[4] L. Gwennap, "MIPS R10000 uses decoupled architecture," *Microprocessor Rep.,* vol. 8, no. 14, pp. 17–22, Oct. 1994.

[5] G. F. Grohoski, "Machine organization of the IBM RISC system/6000 processor," *IBM J. Res. and Devel.,* vol. 34. no. 1, pp. 37–58, Jan. 1990.

[6] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Comput.,* vol. C-30, pp. 478–490, July 1981.

[7] J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures.* Cambridge, MA: MIT Press, 1987.

[8] W. W. Hwu *et al.*, "The superblock: An effective structure for VLIW and superscalar compilation," *J. Supercomputing,* pp. 229–248, July 1993.

[9] T. Ball and J. Larus, "Branch prediction for free," in *Proc. ACM SIGPLAN 1993 Conf. on Programming Languages Design and Implementation,* June 1993.

[10] S. A. Mahlke *et al.*, "Effective compiler support for predicated execution using the hyperblock," in *Proc. 25th Int. Symp. on Microarchitecture,* Dec. 1994, pp. 217–227.

[11] M. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," in *Proc. SIGPLAN'88 Conf. on Programming Language Design and Implementation,* June 1988, pp. 318–328.

[12] J. E. Smith and A. R. Pleszkun, "Implementation of precise interrupts in pipelined processors," in *Proc. 12th Annu. Symp. on Computer Architecture (ISCA-85),* June 1985, pp. 36–44.

[13] G. S. Sohi and S. Vajapeyam, "Instruction issue logic for high-performance interruptible pipelined processors," in *Proc. 14th Annu. Symp. on Computer Architecture (ISCA87),* June 1987, pp. 36–44.

[14] M. Tremblay and P. Tirumalai, "Partners in platform design," *IEEE Spectrum,* pp. 20–26, Apr. 1995.

[15] S. Jourdan, P. Sainrat, and D. Litaize, "Exploring configurations of functional units in an out-of-order superscalar processor," in *Proc. 22nd Annu. Int. Symp. on Computer Architecture (ISCA-22),* June 1995, pp. 117–125.

[16] J. E. Smith, "A study of branch prediction strategies," in *Proc. 8th Annu. Int. Symp. on Computer Architecture,* 1981, pp. 135–148.

[17] B. Calder and D. Grunwald, "Next cache line and set prediction," in *Proc. 22nd Annu. Int. Symp. on Computer Architecture (ISCA-22),* June 1995, pp. 287–297.

[18] A. Seznec, "A case for two-way skewed associative caches," in *Proc. 20th Annu. Int. Symp. on Computer Architecture (ISCA-23),* May 1993, pp. 169–178.

[19] K. Normoyle, Z. Ebrahim, B. VanLoo, and S. Nishtala, "UltraSPARC port architecture," in *Hot Interconnect III Symp. Rec.,* Aug. 1995.

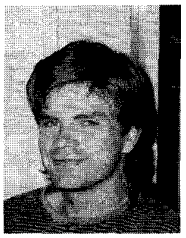
[20] L. Kohn *et al.*, "The visual instruction set (VIS) in UltraSPARC," in *Proc. 1995 Compcon Conf.,* Mar. 1995, pp. 462–469.

[21] C. Zhou *et al.*, "MPEG video decoding with the UltraSPARC visual instruction set," in *Proc. 1995 Compcon Conf.,* Mar. 1995.

[22] D. L. Weaver and T. Germond, *The SPARC Architecture Manual,* Vers. 9. Englewood Cliffs, NJ: Prentice-Hall, 1994.

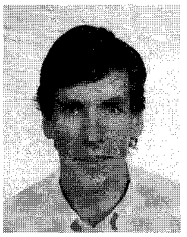
Marc Tremblay (Member, IEEE) received the physics engineering degree from Laval University, Quebec, Canada, in 1984. He received the M.S. and Ph.D. degrees in computer science from the University of California at Los Angeles, in 1991 and 1995, respectively.

He is a Computer Architect involved in the research and development of high-performance processors at Sun Microsystems. Since 1991, his main contributions have focused on the microarchitecture definition and performance evaluation of the 64-b UltraSPARC I and II processors. He has also participated in the definition of SPARC V9 which is the 64-b extension of the existing 32-b SPARC instruction set. His current work relates to improving the synergy between the processor and the compiler, and to including extensive multimedia capabilities directly onto the processor. He is also a member of Sun's Architecture Group, whose charter is to propose and investigate novel architecture features that enhance the performance of microprocessors.



Dale Greenley received the B.S. degree in computer engineering from Santa Clara University, Santa Clara, CA.

He is currently a design manager on a next generation SPARC processor at Sun Microsystems' SPARC Technology Business Unit. His primary responsibility on the UltraSPARC program was the architecture, implementation, and verification of the Load/Store Unit. Previously, he has worked on a variety of CPU programs, most notably on the Integer Unit for the Amdahl/Key Computer Labs ECL Supercomputer and as an original member of the Nexgen x86-compatible processor design team.



Kevin Normoyle received the B.S.E.E. degree from Cornell University, Ithaca, NY, in 1981.

He worked on UltraSPARC's external cache and system interfaces. Before coming to Sun Microsystems, he designed graphics/vector workstations for Stellar and Stardent Computers, and built minicomputers for Data General. He is currently at work on MicroSparc III.