System Design Methodology of UltraSPARC™-I

Lawrence Yang, David Gao, Jamshid Mostoufi
SPARC Technology, Sun Microsystems, Inc.
2550 Garcia Avenue, Mountain View, CA 94043

Raju Joshi, Paul Loewenstein
Sun Microsystems Computer Corporation, Sun Microsystems, Inc.
2550 Garcia Avenue, Mountain View, CA 94043

Abstract - Increasing complexity of microprocessor-based systems puts pressure on a product's time-to-market. We describe a methodology used in designing the system interface of UltraSPARC-I. This methodology allowed us to define the system interface architecture, verify the functionality, perform timing and noise analysis and do the physical board design in parallel. This concurrency permitted rapid implementation of the microprocessor and the system.

I. Introduction

As microprocessors continue to increase in complexity and performance, so do the systems that use these processors. In order to keep up with their microprocessor brethren, system components have been increasing in complexity, clock frequency and level of integration. Because multiprocessing is now widely available on the desktop, it is expected in all future systems, adding even more to the complexity of the verification effort.

Performance is most easily obtained from increased clock rates. That means that increasingly sophisticated techniques must be employed in physical design and electrical analysis: techniques that were once found only in the halls of mainframe and supercomputer design houses.

Increased complexity can mean a longer design time. This increase must occur without impacting the overall time to market of the end product. In order to absorb this increase in design time, the design of various system components must now proceed in parallel with that of other components. Processes must be in place to incorporate design information from other components as they change.

Fig. 1 gives a high-level overview of the concurrency required to execute a program of this complexity. This paper will discuss each of the items shown. It will focus on the design techniques used at the system-level to validate UltraSPARC-I and its surrounding system components. Some tools and techniques that led up to the full-system work will also be discussed.

32nd ACM/IEEE Design Automation Conference ®

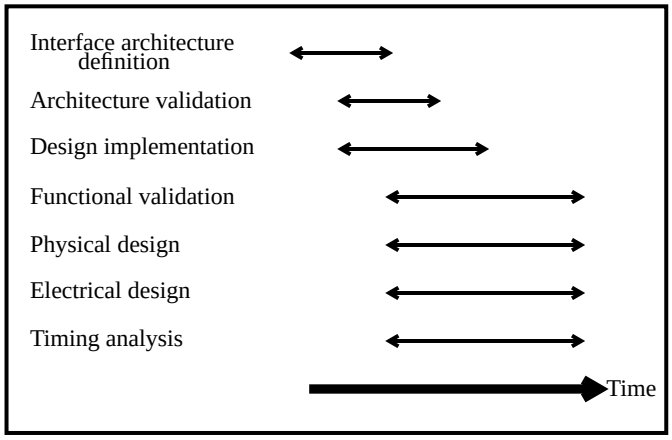

Fig. 1. High-level overview of design methodology, showing concurrency of key activities

II. System overview

UltraSPARC-I is a 64-bit SPARC™ V9 processor with four-way instruction dispatch, superscalar processing and advanced multimedia capabilities [1][2][3][4]. It has a tightly-coupled instruction prefetch and dispatch unit, integer execution unit, floating-point/graphics unit, load/store unit and memory unit. It has external cache control and system interface logic on-board. The chip was carefully designed to maximize system efficiency and promote optimal throughput when executing complex, memory-intensive applications while maintaining full binary compatibility with all existing SPARC applications.

The processor has 16 KB of on-chip instruction cache and 16 KB of on-chip data cache. The processor contains about 5.2 million transistors. The chip is fabricated on an advanced four-layer metal 0.5μm process.

Other components complete the UltraSPARC-I chipset. A pair of data buffers connect UltraSPARC-I to the system and isolate second-level cache activity from the system bus. Each data buffer is a 70K gate gate-array containing logic and datapaths that allow overlapping of operations, resulting in shorter miss latencies and larger bandwidths to and from the system. The external cache is implemented using standard, pipelined 1 Mbit SRAMs organized as 32Kx36 parts.

As is traditional with all of Sun's systems, the core logic of the system was implemented in ASICs [5]. The ASICs

totalled approximately 240K gates of logic and 13K bits of SRAM. Fig. 2 shows a high-level diagram of how the system components are interconnected.

III. System interface architecture definition process

While the CPU design team was still defining the internal microarchitecture, a small group of CPU and system engineers and architects met to define the system interface. This group analyzed memory latency, system memory bandwidth, clock distribution and multiprocessor support.

The challenge was in specifying the interface in enough detail to allow implementation to begin without knowing all the details of other parts of the system (such as the core processor pipeline and I/O requirements), since these other parts were being defined in parallel.

More information regarding the architecture definition process can be found in [6].

IV. Formal Verification

The cache coherence protocol was verified using the Murphi model checker [7] before any Verilog™ HDL register transfer level (RTL) code was written. Very subtle bugs were found, sufficiently subtle that it would not have been noticed until well into multiprocessor simulation. Correcting the error at a later stage could have been costly.

Fig. 3 shows an example of the output from a Murphi run. This run found an error in how the consistency protocol handled writebacks. The detailed state information (in italics) has been simplified for purposes of this paper.

Despite not being a full formal proof of correctness, using Murphi to validate the cache coherence protocol was highly cost-effective. Not only were significant bugs uncovered, but they were found very early, before any simulation model existed. Writing a model at this level of abstraction also provided a deep insight into the working of the protocol. The verification engineer became a designer, actively contributing to the design.

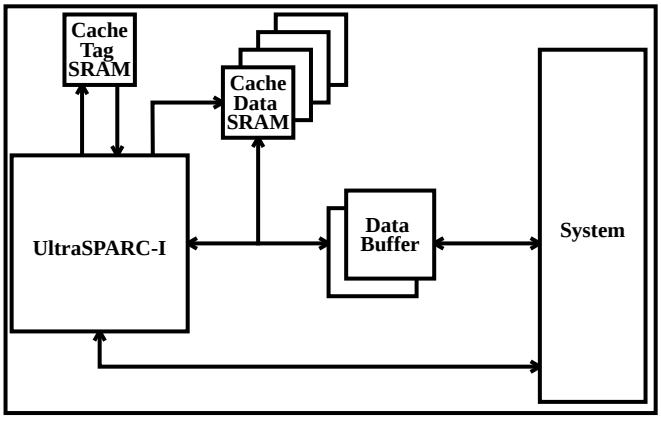

Fig. 2. UltraSPARC-I system block diagram

```
Murphi C++ Verifier, Version 1.4, bfs mode.
Copyright (C) Andreas J. Drexler.

The Murphi description language and verification system
were conceived and designed by David L. Dill, Andreas
Drexler, Alan J. Hu, and C. Han Yang, at the Computer
Systems Laboratory at Stanford University.

Please email bugs/suggestions/comments to
"murphi@theforce.stanford.edu"

This program should be regarded as a debugging aid, not
a certification of correctness.
    Found states: 1,893,517 max, 20.1 bps, 37,213 Kb.
    Active states: 473,379 max, 4.0 bps, 1,849 Kb.

Error in rule Finish_Writeback (line 883):
Inconsistent Writeback Buffer

A violating trace of differences is:

-------- 0 --------
(Initial state)
-------- 1 --------
(Next State)
-------- 2 --------
(Next State)
    .
    .
    .
(Next State)
-------- 6 --------
(Violating State)
No longest trace requested.

Performance:
    8,804 states, 5,669 of them active, 0.5 cps, 56.3s,
331 rps.

State graph:
    Explored part of state graph has 8,804 states and
18,632 multi-edges.

Termination status:
    Error statement executed or assertion violated.
```

Fig. 3. Sample output from Murphi

Murphi has the advantage of being much easier to use than most other model checkers. The cache coherence protocol was written in the Murphi language as a set of guarded rules, describing a non-deterministic state-transition system. Murphi then explores each reachable state explicitly. Clearly, a full-sized description would have too many states, so we scaled down the description to two bits of address, two processors and usually no data at all. Even with this drastic scaling there were about 30 million reachable states, requiring an overnight run on a compute server with 600 megabytes of main memory. However, while the design was buggy, Murphi usually found an error very early, requiring just a few minutes on a workstation.

V. Functional validation

One objective of the functional validation effort was to bring up components of the entire system incrementally and hierarchically (Fig. 4). This process allows design units to be stabilized concurrently before integration into a larger simulation. Another goal was to simulate as much of the system as possible. This goal is very important in light of the time-to-market requirements and complexity of this product.

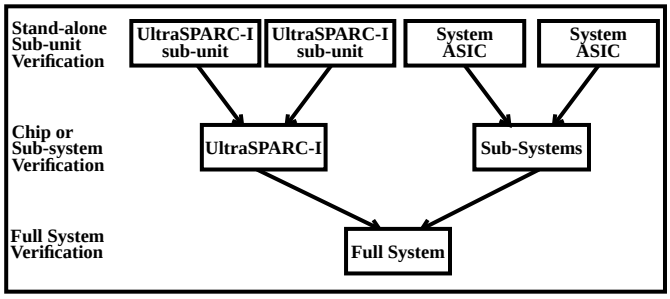

Fig. 4. Hierarchical functional verification

Initially we created stand-alone environments for each of the subunits of the processor and for each of the system ASICs. We implemented both the design itself and all the parts of the design environment in Verilog HDL [8]. Each subunit or ASIC (e.g., integer unit, floating-point unit, etc.) was sanity-checked at a stand-alone level before being integrated into a larger simulation. This effort guarantees a minimal level of functionality so that debugging at the next level of integration can focus on interface problems and not core functionality bugs.

We then integrated the CPU together with simple system behavioral models. Several system behavioral models were used, each one focusing on different system aspects. All the models behave like a complete system so that actual SPARC code could be used as diagnostics (Fig. 5). All the models were capable of not only servicing CPU memory requests but also generating system-initiated traffic, such as interrupts, I/O requests and multiprocessing coherency requests.

One model was tuned to generate simple bus transactions to exercise basic functionality. We later added a random transaction generation capability to enhance the utility of this model. A second model integrated control of bus activity with the SPARC assembly language diagnostic code. A third model emphasized multiprocessing activity.

Thus, these models created an environment that appeared to the CPU model to be the real system design without having the overhead of having to simulate the real design. The combination of the CPU and one of the system behavioral models consumed on the order of 300,000 lines of Verilog code and about 550 megabytes of run-time memory in Verilog-XL.

In parallel to the CPU integration effort, the system design team integrated portions of the system to verify that these interfaces were functionally correct. The CPU designers created a CPU “stub” model to assist the bringup of the system ASICs (Fig. 6). The stub model contained only the system interface and external cache control logic of the processor, along with the SRAM behavioral model and the datapath ASIC netlist. The stub model executed a crude instruction set that allowed an engineer to write a program that caused the stub model to generate and receive different bus transactions.

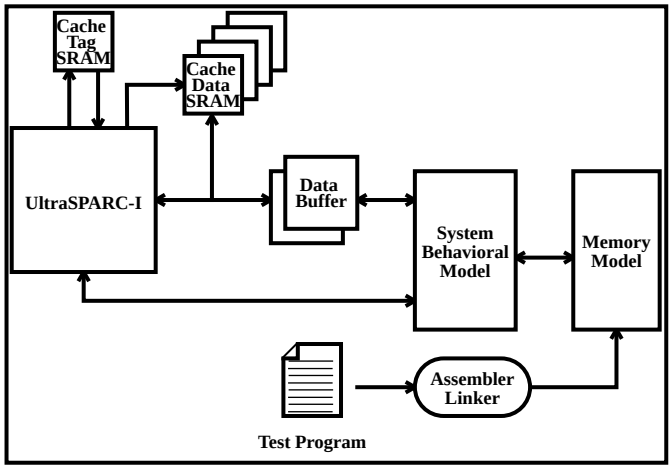

Fig. 5. UltraSPARC-I with system behavioral model

This stub model allowed the system design team to do initial validation of the system design without having to consume simulation cycles debugging the CPU core. Fig. 7 shows how the stub model is used in system validation. The “glue logic” represents behavioral Verilog code that reads and executes the test program. This stub model also allowed designers to generate a large number of system transactions without having to worry about managing a CPU pipeline. Had this effort been done using assembly language tests, a lot of simulation cycles would be lost executing instructions that don’t directly result in system activity, which decreases the amount and density of system transactions. This higher density of system traffic allows bugs in system interface corner cases to be caught quickly.

This stub model environment later evolved into a random testing environment, where a test program was generated randomly and executed on the model. System interfaces can be complex, and the corner cases that cause bugs are hard to predict, so it was found that randomly generated transactions are a good way of uncovering cases that were not foreseen during directed test development.

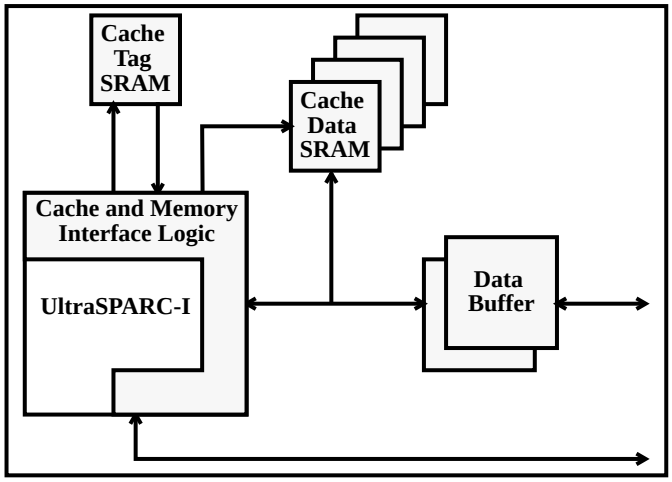

Fig. 6. UltraSPARC-I with “stub” model portions shaded

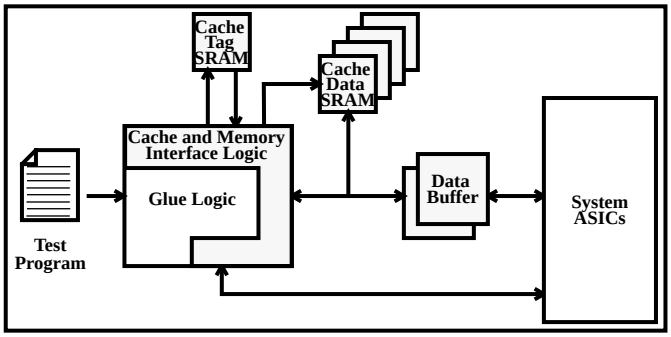

Fig. 7. Application of UltraSPARC-I stub model

Multiple stub models can be instantiated to verify multiprocessing. This eliminates the need to instantiate complete CPU models, thus reducing the simulation memory and compute requirements.

A two-stub multiprocessing environment had about a 240 megabyte Verilog-XL run-time memory image. Each stub was about 20,000 lines of code.

The final test was to simulate the entire system (Fig. 8). There is no substitute for simulating the real design, since only at this level can system-wide problems be found and corrected. The design team created an environment where the full CPU netlist was integrated with key system ASIC netlists. This environment consumed about 730 megabytes of run-time memory in Verilog-XL.

We took advantage of the fact that we were simulating a real computer system by running real programs on the simulation models. SPARC assembly language diagnostics were written, "loaded" into the system memory, and executed, just as in a real system. Trap handlers and page mappings were created to support running these programs. These diagnostics were used to both validate the functionality of the system and to verify that the system performance matched expectations. We also compiled some C programs and ran these programs to check that our new compilers would generate code for UltraSPARC-I that meets our performance expectations.

Finally, when real module and board layouts were completed, logical netlists were extracted and simulated. This step was performed to ensure that the actual board would hook up all the components correctly.

Both Verilog-XL from Cadence Design Systems and VCS from Chronologic Simulation were used in our design. Both tools were used because each offered its own advantages. Verilog-XL has a richer feature set and was particularly useful in the early design phase where we needed to quickly iterate on small design blocks. VCS was more valuable later in the project when the design was more stable and reduction of the memory requirements for our simulations was required. Design Automation's SignalScan and System Science's Magellan graphical waveform viewers were used to help in debugging our simulations.

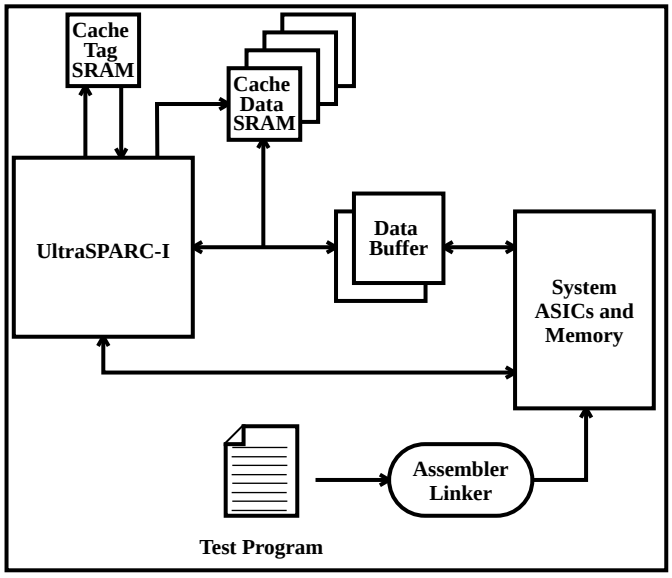

Fig. 8. Full-system simulation environment

A hardware emulation system was also built, integrating all of the microprocessor and system components. This emulation system is described in [9].

VI. Physical Design

Early in the design process we specified the pad assignment and package pinout of each of the components. This early definition was required because of the long lead-time required by the package supplier. The early pad assignment also allowed us to do the internal CPU bus route planning early and to freeze the CPU floorplan. Finally, defining the package pinouts allowed many trial component placements and module layouts to occur to optimize the critical paths on the module.

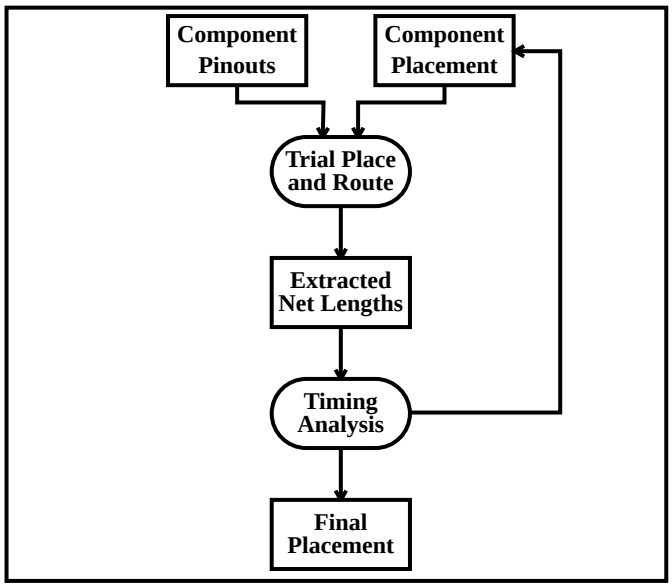

Fig. 9. Module physical design flow

Cadence Concept (formerly Valid's GED) was used for the module schematic entry. Allegro was used for the module layout. Fig. 9 shows the physical design flow.

All the chips used BGA packages. This new technology did not pose a problem for the layout tools since the package footprint looks like that of a 50-mil PGA package.

VII. Electrical Design

Concurrent with the functional and physical design, timing and noise analysis of all the signals between the components was done. (Fig. 10)HSPICE was the primary simulator used for all timing and noise analysis. [10][11].

Circuit designers completed the output driver and input receiver circuits early in the project. These circuits were implemented in test silicon to verify that their performance matched that of their models. Models of the I/O circuits for the external cache SRAM and datapath ASIC were obtained from the various silicon suppliers.

We performed several trials of module component layouts. We then generated approximate net lengths for use in our analysis.

Circuit simulation models were then built using the driver and receiver circuits, package parasitic models, and the module netlengths [12]. Interconnect delays were thus generated and were validated to meet the cycle time requirements.

Clock skew and tester guardband values were budgeted early in the design process and also validated later.

All of the simulation results were then plugged into a spreadsheet (Fig. 11). Two sets of numbers were analyzed for each interface: one set of maximum timings and another set of minimum timings.

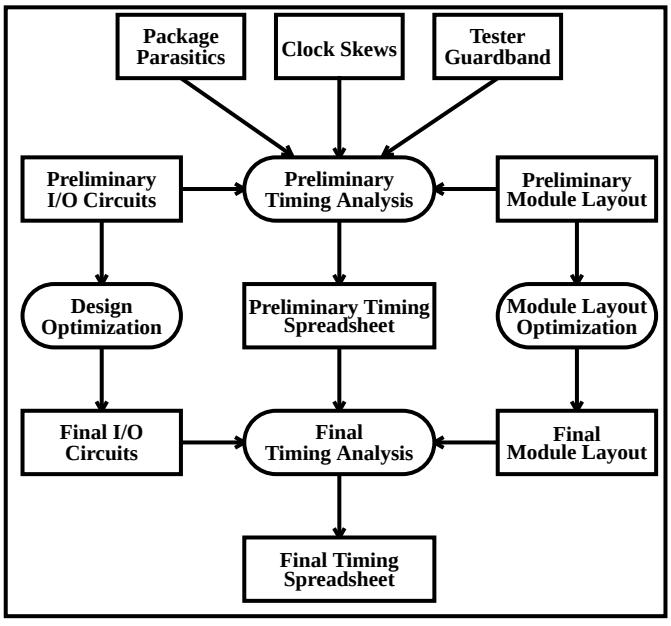

Fig. 10. Module timing analysis methodology

	Max	Min
Clock-to-out	4.1	1.4
Internal clock skew	0.3	-0.3
Tester guardband	0.3	-0.3
External clock skew	0.1	-0.1
Input setup time	0.5	
Input hold time		-0.5
Total	**5.3**	**0.2**

Fig. 11. Timing spreadsheet example

The maximum column uses max output delays and input setup times. The total should be less than the cycle time goal. The minimum column analyzes hold time. It uses minimum output delays and input hold times. Anything that "helps" hold time is a positive number (such as output delay). Anything that "hurts" hold time is a negative number (such as input hold times, clock skews, and tester guardband). The goal is to get a result greater than zero.

At high frequencies it is important to analyze the effects of noise on the system [13]. Our noise simulation used a simulation model of the CPU core, the actual I/O driver and receiver circuits and pad models, a parasitic model of the CPU package, and a parasitic model of the module board.

When modeling the board parasitics we modeled the area under the CPU chip separately from the rest of the board. Because the CPU is a BGA package, there is an array of vias underneath the CPU that results in an array of holes in the ground plane. This "swiss cheese" pattern is difficult model.

Another source of complexity in modeling the board was the fact that although the CPU chip had separate ground pins for the output drivers and the core logic, these pins were connected to the same ground planes on the module. Thus, the module ground plane was a source of noise coupling between the two power domains.

The real CPU pinout has ground pins assigned in an irregular pattern, so the board modeling task was simplified by assuming one core or output ground via for every twelve vias (Fig. 12). Also, instead of an array of holes it was assumed that there was a grid of traces; it was easier to model a set of straight-line traces (Fig. 13).

The module board model was generated using software from Pacific Numeric. This tool created a complex RLC network with over sixty R, L and C components. This network caused us problems in simulation. Using AC analysis, we approximated this network with a three component equivalent circuit model.

The noise is dependent on the switching behavior of the output drivers. Various switching conditions had to be analyzed, including all drivers switching in the same direction, as well as different combinations of drivers switching in different directions.

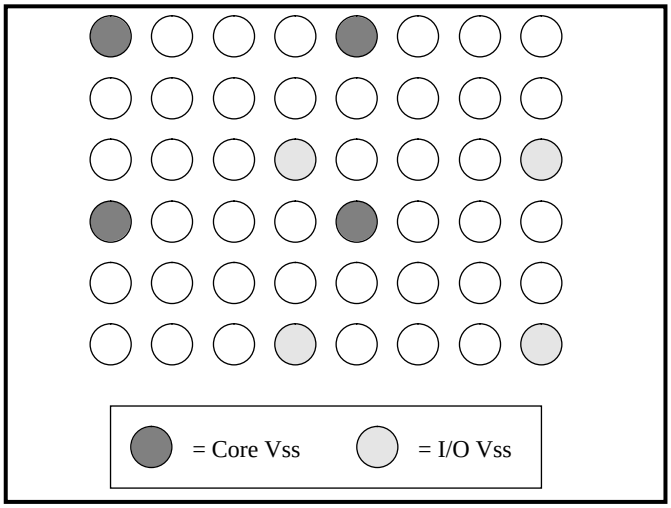

Fig. 12. Approximated layout of core and I/O Vss pins

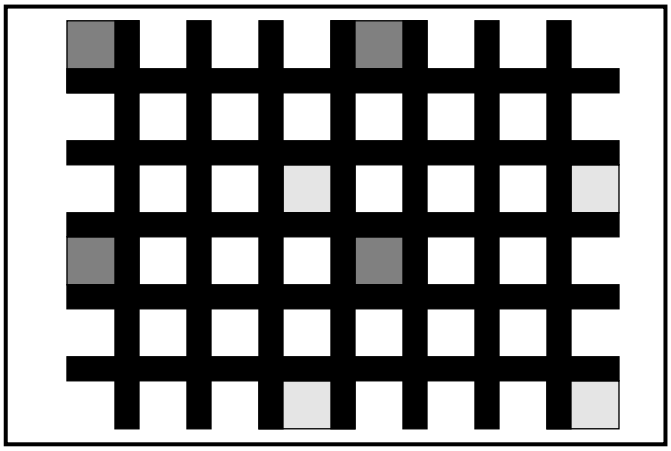

Fig. 13. Reduction of layout of Vss pins into traces

VIII. Results

All of our design efforts resulted in a system bringup that exceeded our expectations. Despite the fact that the system had a new processor and system architecture, a new operating system, a new silicon process and a new packaging technology, the system debug team was able to bring up multi-user Unix in under a week after arrival of first silicon.

One aspect of our methodology that we debated was the need for multiple system simulation environments. Ideally we wanted to focus our simulation time and resources on only one or two environments. However, because different engineers wanted to focus on different aspects of the design, we found that it was impossible for one or two environments to serve everyone's needs. In the end it was beneficial having different people working with different environments. The system interface logic was sufficiently complex that having a diverse approach allowed for broader test coverage.

Acknowledgments

In addition to the authors, several other people were involved in the design and verification of the UltraSPARC-I system interface. Sherri Al-Ashari, Devereaux Chen, Reza Eltejaein, Gary Goldman, G.P. Grewal, Kevin Normoyle and Shaham Parvin were also key contributors to the design and verification of the system interface logic. Christopher Cheng, Sunil Kaul, Mohammad Tamjidi and Leo Yuan helped immensely on the electrical design and analysis.

References

[1] A. Agrawal, "UltraSPARC: A new era in SPARC performance," *Proc. 7th Microprocessor Forum*, Oct. 1994.

[2] L. Gwennap, "UltraSPARC unleashes SPARC performance," *Microprocessor Report*, vol. 8, num. 13, October 3, 1994.

[3] A. Charnas, *et al*., "A 64b microprocessor with multimedia support," *ISSCC Digest of Technical Papers*, pp 178-179, Feb. 1995.

[4] D. Greenley, *et al*., "UltraSPARC: The next generation superscalar 64 bit SPARC," *COMPCON Spring 95 Digest of Papers*.

[5] A. Bechtolsheim, *et al*, "How Sun Microsystems created SPARCStation 1 using LSI Logic's ASIC system technology," in *The SPARC Technical Papers*, New York: Springer-Verlag, 1991.

[6] M. Tremblay, "A fast and flexible performance simulator for microarchitecture trade-off analysis on UltraSPARC." *Proc. 32st ACM/IEEE Des. Auto Conf*., Jun. 1995 (in press).

[7] D. Dill, A.J. Drexler, A.J. Hu, C.H. Yang, "Protocol verification as a hardware design aid," *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp 522-525, 1992.

[8] E. Sternheim, R. Singh, Y. Trivedi, *Digital Design with Verilog HDL*, Cupertino, California: Automata Publishing Co., 1990.

[9] J. Gateley, *et al*., "UltraSPARC-I Emulation," *Proc. 32st ACM/IEEE Des. Auto Conf*., Jun. 1995 (in press).

[10] T.Y. Chou, J. Costentino, and Z. Cendes, "High speed interconnect modeling and high accuracy simulation using SPICE and finite element methods," *Proc. 30th ACMIEEE Des. Auto. Conf*., Jun. 1993.

[11] L.W. Nagel, "SPICE2: A computer program to simulate semiconductor circuits," *Electr. Res. Lab. Report ERL M520*, University of California, Berkeley, May 1975.

[12] H.B. Bakoglu, *Circuits, Interconnections and Packaging for VLSI*, Menlo Park, California: Addison-Wesley, 1990.

[13] H.W. Ott, *Noise Reduction Techniques in Electronic Systems*, New York: John Wiley and Sons, 1976.