Origin System Design Methodology and Experience: 1M-gate ASICs and Beyond

Ásgeir Th. Eiríksson, John Keen, Alex Silbey, Swami Venkataraman, Michael Woodacre
Silicon Graphics Inc.,
Mountain View, CA

Abstract.

The Origin System from Silicon Graphics pushed the complexity limits of ASIC design to levels previously only seen in full custom microprocessors. We describe the methodology used to implement and verify this ccNUMA multiprocessor system. A formal specification, consisting of a detailed, machine readable description of the ccNUMA cache coherence protocol was the corner stone used to manage the complexity of the design. This specification was formally verified and used to automate logic verification. We used a hierarchical approach at all levels to attack the design and verification. We made design decisions to ease verification without compromising system performance. The completion of this system, running at speed, with no bugs in the cache coherence protocol, validates this methodology.

1 Introduction

We describe the methodology that was used to implement and verify the Silicon Graphics Origin 2000, a cache-coherent non-uniform access (ccNUMA) multiprocessor system. The Origin 2000 directory-based shared memory machine [4] shown in Figure 1 consists of nodes of one or two processors, physical memory, directory memory, a node controller Hub, IO Xbar interconnect, IO devices, and a scalable interconnect connecting 2-512 different nodes. The Hub chip is composed of a Crossbar (XB), Processor Interface (PI), Network Interface (NI), IO Interface (II) and a Memory/Directory controller (MD).

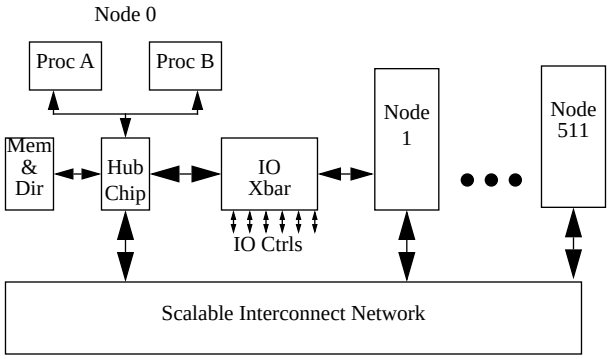

Figure 1 Origin block diagram

The single largest difference between previous generation SGI multiprocessor systems and the Origin 2000 is the ccNUMA architecture. Its cache coherency protocol is invalidation based, and together with the processor [6], supports a sequentially consistent memory model [7]. In order to be independent of a specific network topology, the protocol does not rely on network ordering. This makes verification more difficult by an order of magnitude, as the number of corner cases goes up dramatically. For further detail on the cache coherence protocol, refer to [2, 4].

Implemented in the Hub ASIC, the coherence protocol and support hardware represent the most complex system ASIC designed at Silicon Graphics to date. The design was realized in a five-metal layer, 900 thousand gate standard cell chip running at 100 MHz. The physical design and timing methodologies stressed the limits of available tools.

This paper concentrates on the verification and physical design methodologies of the Hub ASIC.

The paper is organized as follows. Section 2 describes the formal verification of the coherence protocol. Section 3 describes the simulation methodology for unit and system verification. Section 4 discusses the physical design and timing methodologies.

2 Formal specification analysis/verification

The Origin 2000 system is highly distributed and supports 10000's of concurrent memory operations. The cache coherence protocol is therefore inherently complex. It was therefore of crucial importance to formally analyze its correctness. The formal analysis was extremely successful. It found numerous problems that would have been extremely difficult to find with conventional simulation techniques. This included a case where an 18 step sequence of messages led to loss of cache coherency!

We chose **smv** [3] to formally verify the protocol specifications. There are several reasons for this choice (see [1,2] for more details). First, **smv** has been successfully used to verify the specifications of other cache coherence protocols. Another reason is that source code is available for the tool, in case any problems are encountered. Finally, we chose **smv** because it can be integrated with a conventional

project design flow; an important consideration from our industry perspective.

We employ top-down methods to maximize the benefits of formal verification. An overview of the formal analysis workflow is shown in Figure 2. The input to the analysis is a cache coherence protocol specification.

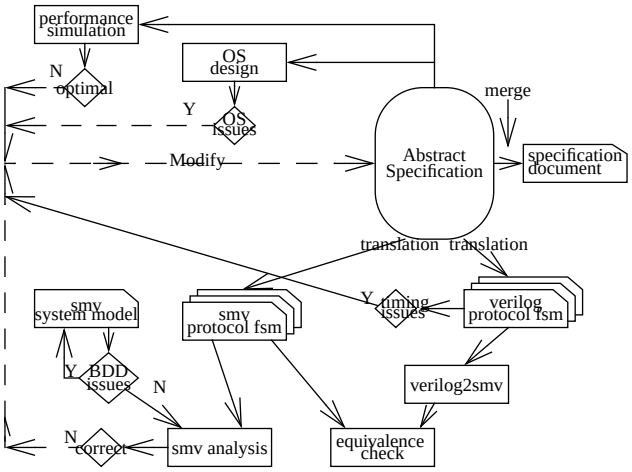

Figure 2 formal analysis workflow

The **smv** model is derived from the design specification. Temporal abstraction is used to minimize the granularity of the time scale, and functional dependency analysis is used to eliminate as many state variables as possible. Finally, we selectively refine the **smv** model, using the RTL implementation. We proceed in this fashion in order to increase the computational efficiency of model checking, and to avoid the state explosion problem [1,2].

A high-level protocol specification consists of a collection of state machine tables, that determine the response to incoming messages in terms of state changes, outputs, and outgoing messages. The tables serve as input to a performance simulation, formal verification with **smv**, verilog RTL state machine generation, simulation table sweeps (see section 3.3), and the protocol document.

During the early phases of the project, protocol design alternatives were evaluated using a performance simulator. After performance evaluation trade-off simulations were completed, the protocol went through numerous revisions. The three primary driving forces behind the changes are the following: operating system (OS) requirements, RTL synthesis timing issues, and protocol problems, uncovered with the formal analysis.

There was exactly one accurate, always up to date, machine readable protocol specification. Having one source for the specification had several important benefits. The first was that the different tools were always working with the same version of the protocol. Another was that it was possible to verify any design changes, e.g. due to RTL timing considerations, with the formal verification tool. Finally, once formal verification found a problem in the protocol, and **smv** verified the proposed fix, the revised version of the protocol was immediately available to the RTL simulation tools.

The goal was to verify the following properties of the high-level specifications: deadlock-free, all coherent read & write requests receive the correct response, there are never unsolicited responses, and no violation of the safety invariants.

If for performance reasons the protocol was implemented with one-hot encoding, we also verified that there is always at most one row activated in a table. It is also important to check the converse condition, i.e. that each row is activated for some state of the protocol.

The following four types of safety properties are verified: expected state machine input conditions, protocol message invariants, protocol state invariants, and a special case of deadlock.

A protocol message invariant, for example, is the property of the cache coherence protocol that a particular processor can only have at most one outstanding request targeting a particular cache line.

A protocol state invariant, for example, is the property that if a particular processor has an exclusive cached copy of a cache line, then no other processor or the I/O merge cache can have a copy of the same cache line.

The important result of the formal analysis is that no protocol problems have been found since the formal analysis was completed.

3 Implementation verification

While formal verification played a crucial role in our verification strategy, traditional simulation-based verification remained an essential activity. This section summarizes our simulation strategy and experiences.

We intentionally designed the Hub chip so as to facilitate the verification effort, and we developed a simulation environment in which we could easily write and execute a very large number of tests. We ran simulations on individual modules within the Hub, as well as on more highly integrated configurations. The following subsections will elaborate on these themes.

3.1 Unit verification

Divide and conquer was the basic approach we took to verify the Hub chip. As the design neatly partitioned itself into five major units- PI, MD, NI, II and a centralized XBar (XB), which passes the messages between the units,

we formed our verification strategy based on the special characteristics of each unit. We created a strategy to verify each of the units through directed diagnostics and with automated table sweep diags. Our goal was to find and fix all the unit level bugs before we moved full steam ahead on the full chip system simulation.

3.1.1 Stubs

To achieve our goal of finding bugs in the unit level, we developed several stubs to make the verification efforts effective and semi-automatic, and so that we could generate some directed random diags easily. Some of the stubs we developed:

- For the processor interface, we wrote a R10000 system interface emulation stub which adheres to the SysAD protocol [6].
- We created a stub to drive the Xbar side of each unit. Basically this stub, core vector generation (CVG), can generate any kind of SN0 protocol messages to be destined to a specified unit. This stub also took advantage of the fact that each of the units interfaced to the Xbar using a set of input and output request and reply FIFOs in a very uniform way. We were able to replace the verilog model for any unit with this CVG stub to send any requests or replies to the unit(s) under test.
- The network interface needed a stub to drive the scalable interconnect network.
- The IO Interface connects to the outside world using an XIO bus and so we wrote a stub to emulate the XIO protocol.

All these stubs were written with higher level task calls, which can be called from the diags directly to make the diag writing simple, readable yet quite powerful.

3.1.2 Directed diagnostics

Most of the directed diags were written in verilog using these above stubs and are made to be self checking. The diags then were compiled into the model and so can be run at anytime using the runtime command line options. Since these diags were written in verilog they could take full advantage of verilog constructs like wait, event and #delays etc.

3.1.3 Table sweeps

We needed to verify that each unit, treated as a black box, would correctly generate all proper transaction responses regardless of any interaction with other transactions. We realized that it would take an inordinate amount of verification effort to come up with directed diags to cover all the messages handled by each unit and the possible interactions of these messages. Also, the fact that the Hub chip implements cache coherency in hardware through a set of protocol tables built in each of the units led us to this new way of verifying these individual tables through a table driven method.

Formal verification methods were used to verify the higher level specification of the protocol table in each of the units and so based on this we could derive a master verilog table to be compared against the tables implemented in hardware in each of the units. The master table specifies in detail what should be the expected state transitions and the output messages, if any, for a given set of input conditions. So the first task of this program is to “walk” every line of the table by setting up initial states and then “hit” it with an input condition and check and make sure that the hardware implementation of the table behaves as expected. Another purpose of this program is to “sweep” multiple transactions to hit various lines of the table and to verify that the unit behaves as specified in the master table in all conditions.

The implementation of this idea was carried out differently for each of the units. The straight forward one being the MD, which only responds to requests coming from the Xbar side of the input queue. In contrast, the PI & II needed to service the requests coming from R10000 and XIO devices respectively, in addition to the ones from their input Xbar queue. This introduced additional complexity, as these units needed some coherency request buffers (CRBs) built into them to keep track of the conflicting coherent traffic in these units. Some of the units gave back door access to setup the initial state of the CRB or Directory states and this was quite useful to achieve the objective to “walk” and hit every line specified in the table. For those units which didn’t provide back door access the initial conditions were set up by using a set of sequences and testing for the expected state at each step of the sequences. For the PI, table sweep was carried out as a set of streams running in parallel with each of the streams consisting of a set of sequences. The number of streams which can run in parallel was limited by the number of outstanding reads supported by the processor to the external agent. For the MD, it was carried out by sweeping all different types of requests against each single type of request. The scope of this was limited by the number of transactions which can be handled in parallel by MD. For II, this was carried out as a set of directed diags running in parallel each targeting different sets of lines in the II protocol table.

On the whole this table sweep verification proved to be quite successful as we could automate most of this using some sets of perl scripts and also made it easier to run sweeps targeting different lines of the table by modifying some command line arguments.

3.1.4 Formal verification of unit sub-blocks

As a part of unit verification, we also formally verified several sub-blocks of units where we felt the directed diagnostic coverage was not sufficient. This included some arbiters, fifos and credit size management FSMs.

3.2 Diagnostic environment

We wrote our directed tests for the II module in a high-level language which we specified and implemented.

Other options we considered were to write our tests directly in Verilog or in some language similar to Verilog for which a convenient macro expansion tool already existed; either way, we would generate Verilog code for compilation with the RTL.

The high-level language provided several advantages over the other options:

- Ease of writing and maintaining tests. The syntax and semantics were more closely aligned to the conceptual level at which we wished to express our tests.
- Portability. The same source code could be executed in two different ways: compilation or interpretation.

Our language is intended to test RTL whose functionality can be described in terms of packets sent to or received from it. The language provides various flavors of two basic commands: *_inject and *_expect. These commands are used to send packets to and receive packets from the RTL, respectively. The * is replaced by specification of a particular stub in the configuration. For each *_inject or *_expect command, the user must specify the contents of the fields of the packet; the definition of what fields comprise a packet depends on the particular stub to which the command applies.

To allow for concurrency amongst events, the language supports optional “NAME” and “AFTER” directives which can be specified with commands. The NAME directive attaches a specific name to a particular command. If a command has an AFTER directive specified for it, then this directive indicates the commands whose completion must precede the execution of the command; these prerequisite commands are denoted by listing their names.

The following example illustrates our language:

```
xt_inject name=xt_iw1 {
  TNUM=6; TYPE=WR_RQWRP; SIZE=DW;
  ... other field values ...
  ADDR=0x3D_0004_0000; DATA=0x3_0004; };

xt_expect name=xt_ew1 {
  TNUM=6; TYPE=WR_RSP; SIZE=DW;
  ... other field values ... };

xt_inject name=xt_iw2 after [xt_iw1] {
  TNUM=7; TYPE=WR_RQWRP; SIZE=DW;
  ... other field values ...
  ADDR=0x3D_0004_00A8; DATA=0x3_D005; };

xt_expect name=xt_ew2 {
  TNUM=7; TYPE=WR_RSP; SIZE=DW;
  ... other field values };

finish after [xt_ew1, xt_ew2];
```

Figure 3 Test written in diagnostic language

In this example, the xt_inject command named xt_iw2 can execute immediately after xt_iw1 has completed; it need not wait for the xt_expect command named xt_ew1 to occur first. These xt_iw1 and xt_iw2 commands both submit XIO write requests to the II RTL. The xt_expect commands named xt_ew1 and xt_ew2 specify the respective responses expected for these requests. The test doesn’t finish until both xt_ew1 and xt_ew2 have happened.

Originally, we wrote a tool which converted our tests from their high-level language representation into Verilog suitable for compilation with the ASIC RTL. However, we later decided to discontinue this approach. Instead of compilation, we opted for interpretation at run-time. We wrote some C routines that parse and execute our tests, and we used the Verilog PLI to connect our C code to the underlying Verilog simulation model. The advantages of interpreting tests at run-time rather than compiling them with the RTL are as follows:

- Faster VCS [11] compilation time (less code to compile).
- Smaller simulation binary executable.
- Quicker development for tests (faster iteration time).
- Ability to run very many tests (sweeps, randoms).

The Hub chip connects to several important communication channels. For example, the R10000 processors communicate with the Hub via the SysAD bus. Likewise, the Hub and its IO devices communicate via the XIO link. We wrote monitors which continuously watch traffic on the SysAD bus and XIO link. If a monitor detects a protocol violation (e.g., a flow control error), it will report the error. These monitors helped us to quickly discover inter-chip communication errors, and provided valuable guidance about the source of these errors.

To gain some insight into traffic patterns during execution of a test, we wrote a tool which generates traces. This tool prints a listing of all packets which travel through the Xbar module. It writes its output to a file in ASCII format which

we can later analyze. When bugs occurred, these traces provided valuable information about activity in the system prior to the time of failure. We also generated complete signal dump files (VCD files) and examined activity on all signals using the signalscan [8] tool. The VCD files were much larger than the Xbar traffic traces and generally covered much shorter time intervals.

3.3 System simulation

The directed and table sweep diags provided the foundation of the functional verification of the Hub, but as we were replacing RTL units with CVG's in a lot of these environments, there was the need to do full chip simulation to make sure the units would inter-operate correctly. We also wanted to make sure the Hub chip could correctly work in a system of multiple Hub chips and router chips.

A single node system simulation consisted of a Hub chip, with the processor interface connected to 2 R10000 MIPS processors, the memory interface having RAVICAD SDRAM models connected, and the IO interface having a XIO stub connected. The R10000 MIPS processors were complete RTL models but were executed as separate co-simulation processes as they run in a proprietary simulation language to MIPS technologies. We used a socket based co-simulation package that had been used on previous generations of systems at SGI.

Stimulus to the system was provided in a number of ways. Firstly, we would compile programs (C or assembly) and then load the object files into the SDRAMS and then have the R10000 MIPS processors come out of reset and start executing this code (a condensed boot setup and state initialization was used rather than executing the software required in a real system). A single node would simulate at 6 cycle/second. These programs would stress the cache coherency by performing operations to cause transaction patterns such as massive false sharing of cache lines, or true sharing. The second stimulus was from the XIO stub which could start up random DMA sequences to the SDRAM memory connected to the Hub. The XIO stub sequences would be self checking, writing and then reading back that data (varying the address, size and data pattern etc.). The processors would also stress the XIO stub with PIO reads and writes, as well as graphics writes (these could be block (16 doublewords) writes, or word writes). Finally, the IO section of the Hub also has a block transfer engine (BTE) for moving large amounts of data around in the system (between memory in one node, or from one node's memory to another). The BTE could be set running by the R10000 processors doing PIO writes to control registers in the Hub.

Once the single node system model was stable, the obvious thing was to run 2 nodes with the NI's directly connected - this allowed us to have 4 processors sharing data covering more arcs in the cache coherency protocol tables. Obviously with the Hub being a large chip, running 2 full nodes together would slow things down. We took advantage of the co-simulation package we had developed at SGI that was originally intended for use connecting together models running in different simulation languages and retargeted it so we could partition the verilog model and have each node simulation running as a separate process. The nodes communicated via UNIX sockets with the Hub's NI sending and receiving data via the sockets. With the current SMP machines that SGI manufactures, we were able to get good scaling of performance by using the multiple processors provided in the system.

The next step was to run more and more nodes, stepping up the node count each time we seemed to plateau the bugs found with the current system size. To manage the complexity of connecting up these large system configurations we used perl to read a configuration file that specified how many nodes and routers were in a system and how they were connected. The perl code would then generate the top level verilog modules and co-simulation interface code required to run the large system models. This turned out to be very flexible and maintainable as we increased the complexity of how we were building and controlling the models.

The Spider router [9] used in the Origin system is a 6 ported router. So for building systems with more than 6 nodes, we would need more than 1 router model. We found that the router RTL model was actually simulating slower than the Hub RTL model as it had a lot of gates directly instantiated in the RTL to meet timing requirements. In order to avoid this bottleneck, we built a virtual router that was actually a piece of PLI code that would take messages from any number of nodes and place them in queues at the destination node. This let us remove the router from the large system simulations and get us back to being restricted on simulation speed by the Hub chip.

The largest system simulation we ran was a 16 node simulation. This simulation actually consisted of 49 UNIX processes that were running together on a 32 processor SGI Challenge server. These were 16 verilog processes for the nodes, 32 processes for the 32 R10000 MIPS processors, and a single parent verilog process that contained the PLI

virtual router for routing data between the nodes. We actually found a bug with this system model!

3.4 Computer resources

We used a compute farm made up of SGI Challenge R4400 200/250MHz servers, supplying a total of over 100 CPUs. In order to make use of this compute resource, we used the LSF queueing tool from Platform Computing [10].

3.5 Bugs discovered in the lab

Despite our intense efforts to thoroughly verify the Hub using formal methods and simulation, a handful of subtle bugs escaped detection until we ran tests on chips in the lab. However, we never found any bug in the formally verified cache coherence protocol.

Of the few bugs found in silicon, the ones that forced us to revise the silicon occurred when there were a lot of independent conditions lined up to create a situation that caused the chip to function incorrectly. These cases involved hard to setup cases in simulation, where lots of state had to build up, with exact back-pressure situations, and precise timings of events. These cases may be best handled by a more formal approach as there are just not enough simulation cycles available to build up all the state required to hit the bug (even when the bugs were known, it was hard to replicate them in simulation). We did use formal verification successfully to help track down problems in the RTL that showed up in the lab.

4 Physical design methodology goals

Our goals for timing, floorplanning and layout were simple:

- provide quick feedback between design and layout
- accurately predict critical paths
- gain early confidence in feasibility of design

While our tool vendors stressed that “deep submicron effects” would require a vastly different design flow, we didn’t find that to be true. Instead, we found that data management is one of the largest problems in designing big chips.

4.1 Tool flow

Our timing/physical design methodology relied on commercial tools for logic synthesis (Synopsys), floorplanning (HLD Systems), and timing (Synopsys/Einstimer). Recognizing early that we’d be stressing all of the tools in new ways, we had 4-way meetings between these vendors and our ASIC vendor, IBM. We wanted to ensure that these tools could handle very large designs cooperatively. We were able to agree on a methodology including Links to Layout, Standard Delay Format (SDF) back-annotation, Physical Design Exchange Format (PDEF), and In-Place Optimization. The multi-hour processing times required by that methodology proved to be too much of a bottleneck. That led us to develop a streamlined methodology using partial SDF, simple Wire-Load Models (WLM) and lots of perl scripts.

For the first time in SGI’s history, we were able to base our sign-off methodology on static timing analysis. We found it difficult to agree with IBM about which WLMs to use for the timing analysis. Our inclination was to use the most accurate ones available, while IBM wanted us to use pessimistic ones. In the end, we agreed to use area-based WLMs sized according to floorplan estimates.

4.2 Timing and synthesis

Very early in the design phase we decided to take a hierarchical approach to timing. We partitioned the logic into about 20 chiplets with between 20-100 thousand gates. The chiplets became our floorplanning blocks. We also recognized that we needed specific timing methodologies for intra-chiplet and inter-chiplet nets.

The 8224 inter-chiplet nets were treated individually as early as possible in the design.

- Timing budget was negotiated and tracked, including time-of-flight across the large die
- Wherever possible, outputs were launched out of the chiplet from a register
- Special WLM was generated for the static timing analysis tools, replacing whole-chip vendor-supplied WLM. This was used before floorplanning, the only time statistics were applied to inter-chiplet nets.
- Tool was written to estimate individual wire lengths based on center-to-center distance between source and destination chiplets. This was useful as a quick check on the floorplan.
- Finally, when ports had been assigned in floorplanning, we extracted SDF for inter-chiplet nets.

The biggest methodology flaw was using the IBM area-based WLMs in combination with Synopsys’ auto-sizing library.

Intra-chiplet nets, on the other hand, were treated statistically all the way through sign-off.

- Initially we used IBM-supplied area-based WLMs.
- Next, we used the floorplanner to generate custom WLMs for each chiplet. This WLM was applied top-down to each synthesis run inside that chiplet.

Designers used Synopsys for all of the logic under their control. We had a dedicated chip integration person do full-chip timing using Einstimer.

4.2.1 Synthesis problems

Our large-chip effort exposed three general problems with the synthesis tool, all related to the quality of results.

- We found it difficult to determine and manage the constraints on the I/Os of the leaf-level modules. It's unfortunate that there aren't any commercial tools to help maintain constraints and timing budgets.
- Synopsys had a difficult time trading off gate sizing and load balancing with IBM's wide variety of drive strengths. Designers were dismayed to find unnecessary back-to-back inverters on their critical paths.
- Synopsys' post-layout optimizations didn't work. We ended up writing some simple perl scripts to resize gates after layout.

Together, these problems led to the overriding problem: excessive hand-instantiation. Under pressure to meet aggressive tapeout schedules, our designers were forced to work around many of these problems by manually designing the logic for timing-critical sections of the chip. The Hub chip was about 30 percent hand-instantiated.

While we did eventually reach our target cycle time, the hand instantiation was time-consuming, hard to get right and slowed down the simulator.

4.2.2 Timing problems

In a new flow, it's important to spend time measuring and improving the correlation between the tools' timing estimates. Early Origin 2000 studies showed gross errors between Synopsys' and IBM Einstimer's timing of high-resistance nets. The errors, in one case as bad as 10X, were due to limitations in Synopsys' delay calculator.

Since we were unable to get enhancements for the delay calculator problem in the time frame of the project, we agreed to avoid the problem altogether. We limited fanout of inter-chiplet nets to four and intra-chiplet to sixteen.

4.3 Floorplanning

Floorplanning provided the crucial link between synthesis and layout. However, we thrashed a lot before we achieved enough accuracy to feel comfortable with the link.

4.3.1 Floorplanning calibration

We needed accurate, quick links in two directions. First, feedback to design needed to be well-correlated with layout but not pessimistic. Second, feed-forward to layout tools had to ensure the correlation.

Pessimistic feedback was immediately rejected by the designers. While it's easy to reduce layout iterations by making draconian WLMs, that forces designers to have to work too hard to make timing. It also leaves performance on the table. Optimistic feedback, on the other hand, could easily result in far too many layout iterations. Designers would tape out an unrealizable chip, and surprises would crop up during physical design.

In the end, we calibrated the HLD/Synopsys feedback path by examining the results of several trial layouts. Path for path, we compared the Synopsys, HLD and layout-derived timing. That allowed us to identify and eliminate the largest discrepancies.

We achieved the best correlation by allowing HLD to autoplace the drivers and receivers of inter-chiplet nets, then feeding those choices to the layout tools. By constraining the layout tools to use these placements, we made sure that inter-chiplet nets wouldn't offer any surprises.

We further improved the correlation by preplacing the components of the most important datapaths. This had the attractive side benefit of improving routability and compactness. Most importantly, though, it improved the predictability of the physical design process.

4.3.2 Logical vs. physical hierarchy

One of the difficult project decisions was how to structure the hierarchy in physical design. Since we were already overburdened with inventing new flows, we decided to take the safe route. We decided that the logical and physical hierarchies would be almost the same.

We felt nervous with our vendors' assurances that PDEF would allow us to bridge disparate hierarchies. We also wanted to achieve quickest possible turnaround between synthesis and floorplanning.

Keeping logical and physical hierarchies the same meant that we didn't need PDEF in the flow. That kept the turnaround time as short as possible. It also made it easier for the logic designer to debug problems stemming from the physical domain.

One unforeseen problem in the link between HLD and Synopsys involved the SDF for inter-chiplet nets. SDF stores interconnect information as point-to-point delays between instance pins. When you read SDF into your timing tool, it attaches the delay to all the pins. Unfortunately, we often found that, because of the lengthy loop through floorplanning, the logical design had changed and no longer corresponded with the SDF. Even though the point-to-point connections were the same, resynthesizing the blocks invariably changed the instance names. That meant the SDF sometimes was unable to attach delays to now-missing instances, and sometimes the delay was attached to the wrong instance.

This is but one example of the difficult data management problems we faced on the Origin project. We constantly struggled with data consistency between logical and physical design. We created disruption in the design verification environment by asking designers to check in updates for physical design before they could be logically verified. That broke the "top of trunk" design which a lot of design verification people were trying to simulate. Towards the end of the project we gained control over these issues by mandating well-publicized snapshots of the netlists.

5 Conclusions

The Origin 2000 Hub chip is the largest and most complicated ASIC ever designed at SGI. To successfully and efficiently accomplish the verification and physical design tasks, we applied the following innovative techniques:

- Formal verification of the cache coherence protocol and portions of the Verilog RTL.
- Table-driven verification of individual modules.
- Module designs which facilitate verification.
- Interpretive simulator interface for running tests.
- Chip designs which facilitate physical design (i.e., recognition of the physical effects of logic design).
- Hierarchical ASIC design methodology.

6 Acknowledgments

In addition to this paper's authors, the following individuals all made substantial contributions to the verification of the Hub and router chips for the Origin 2000 system: Bob Alfieri, Pat Conway, Bulent Dervisoglu, Ben Fathi, Martin Frankel, Dick Hessel, George Kaldani, Yuval Koren, Viranjit Madan, Todd Massey, Dawn Maxon, Curt McDowell, Ken McMillan, Mike McNamara, Chuck Narad, Peter Ostrin, Alex Petruncola, Rich Weber, Steve Whitney and Eric Williams.

Furthermore, the following individuals contributed to the physical design efforts: Joanne Allen, Stan Bailes, Thom Derenthal, Dave Harmon, Dave Koller, Ron Nikel, Steve Padnos, Dave Parry, Rick Paul, Tuan Tran, and Gutrum Wolski.

References

[1] Asgeir Th. Eiriksson and Ken L. McMillan, Using formal verification/analysis methods on the critical path in system design: A case study. In *Proceedings of Computer Aided Verification Conference*, Liege Belgium, LNCS 939, Springer Verlag, 1995.

[2] Ásgeir Th. Eiríksson, "Integrating Formal Verification Methods with A Conventional Project Design Flow", Proc. 33rd ACM/IEEE Design Automation Conf., 1995.

[3] K. L. McMillan, "Symbolic Model Checking", Kluwer Academic Publishers, 1993

[4] Jim Laudon, and Dan Lenoski, "System Overview of the Origin 200/2000 Product Line", COMPCON '97.

[5] Steve Whitney, Ken Jacobsen, et. al., "The SGI Origin Software Environment and Application Performance", COMPCON '97.

[6] R10000 Microprocessor User's Manual, http://www.mips.com/products/r10k

[7] S.V.Adve, "Designing Memory Consistency Models For Shared Memory Multiprocessors", Ph.D. Thesis, U of Wisconsin-Madison, 1993.

[8] Signalscan, Design Acceleration, http://www.designacc.com

[9] Mike Galles, "The SGI SPIDER chip", Hot Interconnects Symposium IV, 1996.

[10] LSF, Platform Computing, http://www.platform.com

[11] VCS verilog compiler, http://www.chronologic.com