Invited paper presented at the 2nd Workshop on Complexity-Effective Design, Goteborg, Sweden, June 2001

Managing Complexity in the Piranha Server-Class Processor Design

Luiz André Barroso, Kourosh Gharachorloo, Mosur Ravishankar and Robert Stets
Western Research Laboratory
Compaq Computer Corporation
Palo Alto, CA 94301

Abstract

High-end microprocessor designs have recently been incorporating increasingly advanced features, such as larger issue width and speculative out-of-order execution, which are targeted at further extracting instruction-level parallelism from programs. The added design complexity introduced by such mechanisms has led to an alarming increase in design cost and time-to-market for next generation designs. Although these mechanisms have improved performance of some applications, many important commercial workloads, such as online transaction processing, have not benefited significantly from them.

In response to these trends, the Piranha project set out to address commercial workload performance requirements while managing overall project complexity. In this paper, we discuss the Piranha architecture and the project's novel design methodology, which together enabled the design to be brought to the point of a physical prototype by a team of less than 20 people working for little over a year.

1. Introduction

Recent trends in high-end processor design have been towards aggressively exploiting instruction level parallelism (ILP), largely by increasing issue width and implementing speculative out-of-order execution. Such trends have benefited some applications, such as those modeled by the SPEC benchmarks [15], but at the cost of significant additional design complexity. While the added complexity naturally increased the architectural and logic design time, its most serious impact has been felt in the verification and physical design phases, which already constitute the most time intensive tasks in such efforts. The additional operational complexity of the new designs directly drives additional verification requirements, and the associated physical implementations are larger and more complicated, thus increasing the physical design challenges.

Interestingly, not all applications benefit equally from these complicated designs. Recent research [2, 3, 4, 12] shows that complicated mechanisms for exploiting ILP may have limited benefits in important commercial workloads such as online transaction processing (OLTP). OLTP applications are dominated by short-lived transactions that are memory intensive and exhibit little instruction-level parallelism. These applications do, however, have abundant thread level parallelism.

In the Piranha project [2], we sought to build a processor that would provide superior performance for commercial workloads with project complexity appropriate for tens of people, rather than the hundreds of people typical in today's commercial processor projects. Complexity is managed in Piranha both through architectural choices and design methodology features. Piranha uses a novel architecture, a chip multiprocessor (CMP) based on eight single-issue, in-order processor cores, that simultaneously addresses both commercial workload performance requirements and project complexity goals. The relatively simple processor cores sacrifice single-threaded performance, however the collection of cores, tied together by a tightly coupled memory hierarchy, provides excellent multi-threaded performance. The relatively simple cores also help manage verification overhead, and inherent module replication in the architecture, *e.g.* the processors cores and their caches, reduces the count and size of unique modules in the physical design phase. Our system architecture further addresses project complexity by reusing logic modules in processing and I/O nodes, thus reducing the amount of unique I/O path circuitry that needs to be designed, implemented, and tested.

In addition to architecting for lower design complexity, we further tackle design and verification complexity through our design methodology. We have developed a novel C++-based ASIC design flow that drives a very high performance logic simulator and industry standard ASIC design tools. The high performance simulator enables a very efficient test plan based on testing of the integrated chip, while the ASIC methodology automates much of the low level physical design work and provides a path to fabrication by a third party. Together, this methodology and the architectural choices outlined above allowed us to very effectively manage overall project complexity without compromising on performance with respect to contemporary commercial processor designs.

After over a year of design, verification, and implementation work, the Piranha project was on schedule to deliver its first physical prototype in mere months, when

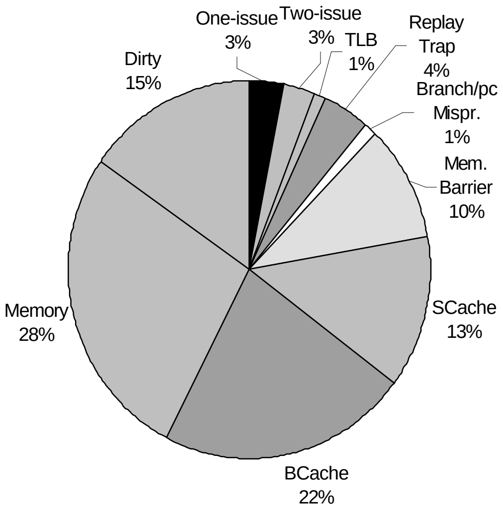

FIGURE 1. Normalized execution time breakdown of an OLTP workload.

budget constraints prevented us from going ahead with fabrication.

In this paper, we will discuss the Piranha project and our efforts to manage overall complexity. The following section will discuss the Piranha architecture and its motivations. Section 3 discusses the project's logic verification phase, including the ways that our methodology helped manage verification overhead. Section 4 describes the obstacles encountered in building a high performance processor using an ASIC process. Finally, Section 5 summarizes the most relevant aspects of our experience with Piranha.

2. Architecture

The Piranha architecture was driven by detailed and extensive performance analysis of database workloads. This analysis shed light on the shortcomings of conventional processor architectures when running OLTP workloads and led to alternative architectural ideas. Our choice of architectural features obviously has a profound effect on the verification and physical design overhead. In the remainder of this section, we describe the aspects of OLTP workloads that are most relevant to processor and memory system design. We next discuss the Piranha architecture in more detail, focusing on the important decisions that balance performance requirements versus project complexity.

2.1 OLTP Performance Characterization

Current database workloads are dominated by processor and memory system performance issues. Recent studies [2, 3, 4, 12] have shown that OLTP workloads exhibit little instruction level parallelism and are largely bound by memory latency.

Figure 1 illustrates the execution time breakdown of an OLTP workload that models the industry standard TPC-B benchmark [16]. This figure is based on an earlier study [4] that examined the memory system performance of the same OLTP workload running on Oracle database software and an AlphaServer 8400 server with eight Alpha 21164 processors. In the figure, execution time is split into cycles spent in single and double instruction issue, TLB misses, replay traps, branch mispredictions, memory barriers, secondary cache hits, board cache hits, memory accesses and dirty misses. As can be seen from the figure, 78% of the execution time is spent stalled on the memory system. With the low percentage of cycles where multiple instructions could be issued and the high memory stall time, it became clear that the current processor trends were not well addressing OLTP performance requirements. Subsequent studies by our group and others confirmed this observation [10, 12]. Fortunately, other processor design trends, which do not significantly increase complexity, do improve OLTP performance. The Alpha 21364 [1] processor takes advantage of increasing transistor density to integrate a second-level cache, a memory controller, cache coherence engines, and a router on to the processor chip. The result is a reduction in chip boundary crossings and, correspondingly, lower memory latencies and higher bandwidths. A simulation-based study [3] showed that such integrated design outperformed a comparable system based on the conventional design by a factor of 1.5x.

Another recent architectural technique that shows promise for improving OLTP performance is Simultaneous Multithreading (SMT) [6]. While SMT addresses commercial workload requirements by exploiting thread level parallelism, it does not address the issue of design complexity as it based on a very complex processor core.

In Piranha, we chose to use a chip multiprocessing as an alternative for effectively exploiting the abundant thread-level parallelism in OLTP workloads. The following section discusses how our design addresses both performance and complexity simultaneously.

2.2 Piranha Architecture

In this discussion, we limit ourselves to describe only the features of the Piranha architecture that concern design complexity issues. A more detailed description of the architecture is available in an earlier paper [2]. Figure 2 shows a block diagram of a Piranha processing node. The node consists of eight Alpha processor cores (CPU), each of which has its own private L1 instruction (iL1) and data (dL1) caches. The L1 caches are connected to the rest of the system via the Intra-Chip Switch (ICS). A logically shared L2 cache is connected to the ICS. The L2 is actually

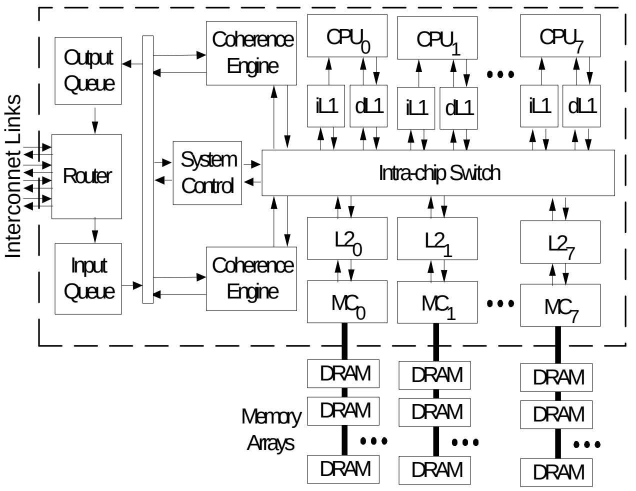

FIGURE 2. Block diagram of a Piranha processing node.

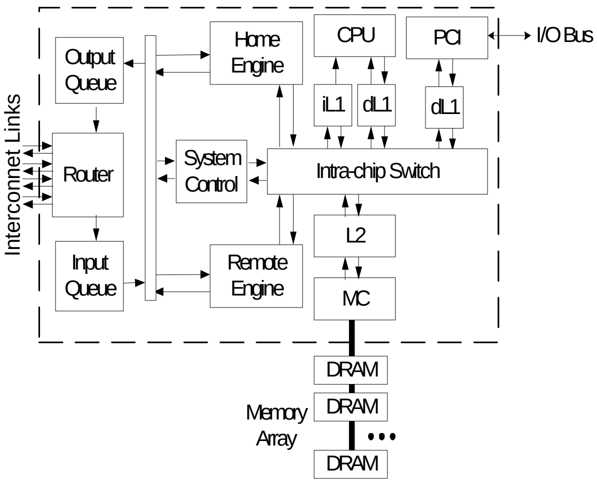

FIGURE 3. Block diagram of a Piranha I/O node.

split into eight banks, with a particular bank selected by the low order three bits of the memory block address. Each L2 bank is connected directly to its own Memory Controller (MC), which in turn connects to its own Double-Data-Rate SDRAM array.

The Home and Remote Engines, collectively called the Coherence Engines, are responsible for handling coherence operations for memory blocks whose home node is the local node or a remote node, respectively. Both of these modules are connected to the ICS, to communicate with the local modules, and to the Input and Output Queues, to communicate with other nodes. The Input and Output Queues both interface to the Router, which in our initial prototype supports a simple ring-based topology.

Finally, the System Controller is responsible for maintenance-related functions such as initialization, interrupt distribution, and exception handling.

The Piranha design includes a very optimized memory hierarchy. In order to maintain cache coherence, Piranha uses a two-level scheme. At the chip level, the L2 is responsible for maintaining coherence within a node. The L2 maintains a set of duplicate L1 tags, and uses this information like a directory to implement an invalidation-based protocol. At the system level, coherence across nodes is maintained by the Home and Remote Engines. This inter-node protocol is also an invalidation-based directory protocol. This directory is stored in “extra” main memory ECC bits, which are freed up by calculating ECC across a memory chunk that is larger than normal. Overall cache coherence is enforced through cooperation between the L2 cache and the coherence engines.

Figure 3 is a block diagram of an I/O node, the second piece of a full Piranha system. The I/O node is actually a scaled-down version of the compute node, with one processor core instead of eight. The I/O node also has a special PCI Interface (PCI) module that integrates the PCI subsystem into the memory hierarchy. Piranha treats the I/O subsystem as a first class citizen of the memory hierarchy, and as can be seen, the subsystem interfaces directly to a standard L1 data cache. This design helps to manage project complexity, since both verification and physical design overhead are reduced because I/O subsystem introduces only one new module: the PCI Interface. Also, as the I/O node is truly a scaled-down compute node, it can serve as an excellent prototype vehicle. Our plan was to focus our verification efforts on the processing node since it provided a more stenuous test of the coherence mechanisms, but to use the I/O node as our first physical prototype. This approach is reasonable since the processing and I/O nodes share so many basic components.

Architectural Decisions. Given the behavior of our target workload, we were able to drastically reduce design complexity by abandoning ILP techniques, and using simple single-issue, in-order processor cores. Another significant simplification was the use of blocking L1 caches. This simplified not only the L1 but the overall memory system design by reducing buffering requirements and making it simpler to keep track of memory transactions throughout the system. The use of blocking L1 caches resulted in only a small performance penalty given that commercial workloads tend to have very few outstanding misses at a time, even when using aggressive out-of-order cores [12]. Further simplification of the core was obtained by not implementing multimedia instruction set extensions, and by employing a straightforward floating-point unit. Server-class commercial workloads have no use for multimedia instruction set extensions and rarely require any floating-point computation.

The general approach of using simple modules and replicating them was employed not only in the processor cores, but throughout the design. The L2 cache, although logically a single shared cache, is implemented as eight separate modules. The logical L2 must handle concurrent local requests from up to sixteen L1 caches and the Coherence Engines. Rather than designing a complicated, monolithic L2 module capable of handling a large number of concurrent requests, we instead broke the L2 module into a simpler design capable of fewer concurrent requests, and then replicated this module to provide the necessary throughput. Since L2 requests can be handled independently based on the target address, this simpler design incurred very little logic overhead, and therefore served to reduce the verification and physical design requirements.

Replication is again used in the design of the Coherence Engines. The Home and Remote Engines are roughly the same design except for a few isolated differences due to the Home Engine’s need to manipulate directory entries. There is however another aspect of the Coherence Engine design that arguably had a more profound effect on project complexity. Rather than embedding the entire protocol implementation in the hardware, the Coherence Engine was designed as a programmable engine, operating on a custom microcode. This design decision forces the separation of the protocol mechanisms, which can be represented as a set of discrete operations, from the protocol policy, which can be represented by a set of inter-related, complicated state machines. The former is much easier to express and build into hardware, while the latter is more error prone and is most easily expressed and maintained at a higher level in the microcode. While a programmable engine is generally slower than a corresponding hard-wired one, our design included several optimizations that effectively reduced the gap to a point where the workload performance was not significantly impacted.

A final important aspect of our design is very straightforward, yet requires a good amount of design discipline to enforce. The module interconnections in the above diagrams are the *only* interconnections in our implementation. By design, we required that each module have a very strictly defined interface and that all outputs of modules were generated directly from flip-flops. This choice helps manage verification and physical design complexity by making modules easier to isolate and by limiting wire lengths mainly to the enclosing modules.

To summarize, the Piranha architecture held both performance and complexity as primary design criteria. The general themes underlying the design choices were to leverage replication and re-use and to pay strict attention to level of complexity to be placed in hardware. It was possible to make informed design trade-offs because of the decision to focus on a single, well understood workload.

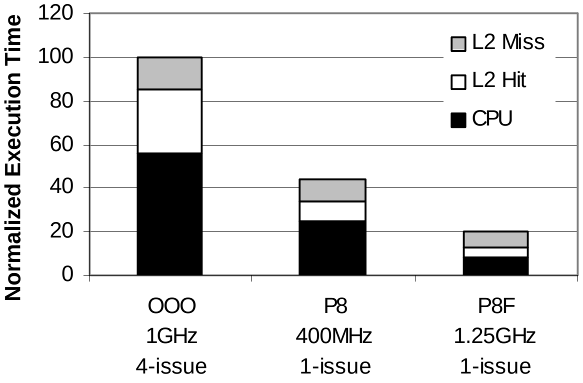

FIGURE 4. Normalized execution time breakdown of a 4-issue, speculative out-of-order, 1GHz processor, a single 400Mhz Piranha chip with 8 single-issue, in-order processor cores, and a single 1.25GHz Piranha chip.

Piranha Performance Evaluation. In the remainder of this section, we will briefly highlight the simulation results of the Piranha architecture. An earlier paper [2] provides a much more complete treatment. The experiments were performed using the SimOS-Alpha simulator, which is an Alpha port of the SimOS [13] simulator. The workload under simulation is again based on the TPC-B benchmark running on an Oracle database.

Figure 4 shows the execution time breakdown of a next generation conventional processor design and two Piranha-based systems. Execution time is broken down into CPU busy time, time spent on accesses that hit in the L2 cache (L2 hit), and time spent on accesses that miss in the L2 cache (L2 miss). The conventional processor design, called OOO in the figure, is a 4-issue, speculative out-of-order processor clocked at 1GHz. The P8 entry in the figure represents a single Piranha chip (with eight processor cores) running at 400MHz. This clock speed is the target speed for the ASIC implementation of our chip. The P8F entry is again a single Piranha chip, but this time built with a full custom design process and clocked at 1.25GHz. This clock speed is estimated on current processor clock rates and the perceived difference in complexity between two designs. One should note that this is actually a per-chip comparison -- the conventional processor design has only one complex processor core, while the Piranha design has eight simple cores.

As can be seen in the figure, the Piranha ASIC and full custom designs outperform the conventional processor design by approximate factors of 2x and 5x, respectively. The performance improvement is driven by the fact that Piranha can better exploit the abundant thread level parallelism. As one thread stalls on a cache miss, other threads on other processor cores can be making progress. Overall, the Piranha architecture addresses OLTP performance requirements so well that even a relatively slow clock speed offers potential for dramatic performance improvement.

3. Logic Verification

In planning our verification strategy, we believed the memory hierarchy, specifically the cache coherence mechanism, would be the most time consuming verification task. The memory hierarchy includes most of the modules of the chip (the L1s, the L2, the Memory Controller, the Coherence Engines, and the network I/O modules), and its actions depend heavily on the global system state, namely the contents of the various caches and the temporal combination of in-flight transactions. To be robust, a design must be tested over a multitude of subtle module interactions and memory access races.

The Importance of Simulation Performance. On top of this inherent verification complexity, our memory hierarchy design was large and so would not simulate efficiently in typical Verilog-based logic simulation environments. Verilog uses an event-driven paradigm to model the implicit parallelism of hardware. Perhaps due to this paradigm, even the best-of-class Verilog simulators are known to run slowly on most large designs. Designers compensate for this lack of simulation speed by relying heavily on isolated sub-unit testing. This approach involves constructing bus functional models (BFMs) to act as test harnesses around isolated units. The required effort to build, validate, and maintain these BFMs, along with the effort required to coordinate system-wide testing at an isolated sub-unit level, greatly increases the overhead and complexity of a design's verification. In our particular case, an isolated sub-unit testing approach would have been extremely difficult, error prone, and labor intensive given the complexity of the interactions among the modules that make up our memory hierarchy.

We believed our test plan had to focus on testing the fully integrated memory hierarchy, and that a *cycle-based* simulator could provide the necessary level of performance. Cycle-based simulators leverage the fact that in synchronous designs, such as ours, signal transitions occur only at well defined points, such as the transition of a clock or a few asynchronous signals, *e.g.* Reset. A cycle-based simulator evaluates signal logic only at these well defined points, thereby eliminating the need for costly event-driven mechanisms. We wrote our simulator in C++ and used C++ to model cycle-accurate, Register Transfer Level (RTL) models of our memory hierarchy modules. To further simulation efficiency, we separated the simulation of the memory hierarchy and the Alpha processor core, as the testing requirements of each of these logical modules are distinct, and importantly the design size and verification requirements of the Alpha Core were smaller relative to the memory hierarchy.

Our environment consists logically of two simulators: the *System* and the *Alpha Core*. The former contains the entire memory hierarchy, and hence basically all of our system, and was fully implemented in C++. The Alpha Core simulator, on the other hand, is Verilog-based and it leverages a number of standard components used in the development of commercial Alpha processors. The rest of the section discusses these two simulators in further detail, and also the viability of using C++ as a hardware description language. Once the components of our simulation environment have been described, the section will conclude with an overview of our test plan.

3.1 System Simulation and Methodology

Driven by the need for high performance, our system simulation employed a novel methodology based on a C++ RTL model of the design. These models, along with some coding conventions and library support, allowed us to build a very fast logic simulator. We then employed a machine C++-to-Verilog translator to provide a bridge to industry-standard ASIC synthesis and physical design tools.

Next, we will discuss our approach to using C++ as a hardware description language (HDL), followed by an overview of our C++-to-Verilog translation process. Finally, we will discuss the full system simulator environment.

C++ as a Hardware Description Language. C++ is an inherently sequential language, and as such may not seem to be a good choice to model the implicit parallelism in hardware. However with some insight into the synchronous hardware designs, C++ can be used as very reasonable HDL. The key insight is that cycle-based simulation eliminates the need for event-based language constructs, which are the fundamental difference between C++ and Verilog. C++ can be used to construct a cycle-based simulator where each module is simply modeled as a function. Still, however, the sequential nature of C++ can cause problems inside of functions and across the inputs and outputs of functions. Without special care, a synchronous variable could be updated inside a function to its next-state value, and then read, in error, later in the function or another module's function.

To solve this problem, we required that all inter-module synchronous variables were modeled by a special "Signal" class that overrides all of its access and assignment classes, and then implements a type of delayed assignment. Any assignment to a variable of this class is buffered internally and then takes effect only at the next clock transition. As a lower overhead alternative for intra-module synchronous signals, we required all assignments to be done to a manually constructed shadow variable, by convention named by the target variable name appended with “_ns”. At the end of the function, all _ns variables are copied into their counterparts.

The Signal class and our _ns convention enabled us to build an efficient cycle-based simulator with a manageable degree of programming complexity. We also decided that our modules would be coded at a cycle-accurate, structural RTL level. We believed this level of specification would facilitate our original plan to perform manual Verilog translations, but as it turned out, the choice actually enabled a machine translation approach.

Machine C++-to-Verilog Translation. Our initial plan was to manually translate C++ to Verilog after the C++ models had become stable and entered a long test phase. Our module representations, however, grew larger (~5,000 to 10,000 lines of C++ code) than expected. Consequently the feasibility of maintaining separate C++ and Verilog code bases became a concern. We chose instead to use CLevel’s System Compiler [5] to perform machine C++-to-Verilog translation.

The System Compiler is a self-contained tool that translates standard ANSI C/C++ to Verilog. The tool’s major coding requirement is that in each module function, the asynchronous logic is separated from the synchronous logic. Fortunately, by virtue of our Signal class and _ns convention, our code already adhered to this convention. It was therefore a relatively straightforward process to integrate the tool into our flow, even though our C++ RTL had been developed without considering the tool.

The PS1 Simulation Environment. Our C++-based design methodology [9] was implemented in the Piranha Simulator (PS1) environment. The tool flow of this environment is shown in Figure 5. The entire flow is based on the C++ RTL models of our design modules. These models are compiled and linked with some support routines to build the PS1 logic simulator. The resulting simulator is a factor of 50 times faster than a comparable Verilog-based simulator [9]. As a bridge to the physical design tools, the RTL models can be passed through the CLevel tool to create corresponding Verilog versions. In order to ensure these models are suitable for physical design, they must be validated against the master C++ models. The PS1 environment includes a special version of the simulator suitable for this task.

The PS1V simulator can be built with an arbitrary mix of C++ and Verilog versions of the modules. The simulator is capable of running either the C++ or the Verilog version of a module, or it can *co-simulate* both versions of a module. The co-simulation facility automatically compares module outputs on a cycle-by-cycle basis in order to establish correspondence between the two versions. We validated the Verilog translations using co-simulation.

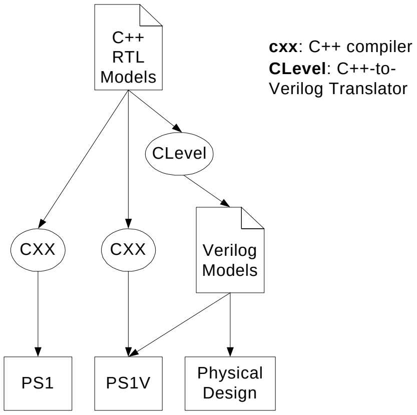

FIGURE 5. The PS1 simulation environment. The C++ RTL models drive a high performance logic simulator (PS1), a mixed C++ and Verilog simulator (PS1V), and industry standard physical design tools

As alluded to earlier, the PS1 environment is intended for verification of the memory hierarchy implementation. As such, PS1 accurately models the entire system, except for the Alpha Core. The Alpha Core contains a large amount of functionality that is largely de-coupled from the memory hierarchy. PS1 focuses its simulation cycles on the memory hierarchy, and replaces the Alpha Core model with a low overhead interpreter driven by a custom test language. The interpreted test language had support for co-routines and the ability to monitor internal signals in the memory hierarchy. We used these capabilities to construct self-checking memory access routines, which were the building blocks of our memory hierarchy tests.

In summary, the Piranha memory hierarchy follows a novel C++-based ASIC design methodology that enables a high performance logic simulator and supports the use of industry standard Verilog-based physical design tools, all while being driven from a single C++ code base. The high performance simulator enables a test plan based on integrated testing that considerably reduces the overhead in testing our memory hierarchy, the critical path of our verification plan.

3.2 Alpha Core Simulation

The size of the Alpha Core design was small enough to simulate efficiently in Verilog, and it could be verified with only limited memory hierarchy support. We therefore decided to follow an established path by implementing the Alpha Core RTL model in Verilog and by using a separate simulation environment that heavily leveraged tools from

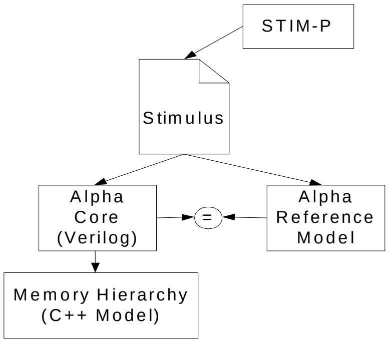

FIGURE 6. Alpha Core simulation environment. Stimulus files drive our Alpha Core Verilog model, with the internal state compared against a reference model on a cycle by cycle basis.

the Alpha 21364 verification effort.

The components of our Alpha Core simulator are shown in Figure 6. The simulator is driven by a stimulus file consisting of valid Alpha instructions. This file can either be manually constructed or automatically constructed by STIM-P, a special pseudo-random test generator from the Alpha 21364 project. The stimulus drives both our Alpha Verilog model and a special Alpha Reference model built by the Alpha 21364 team. As the figure shows, the Alpha Verilog model was backed by our C++ model of the memory hierarchy.

The Alpha Reference model is a high level, behavioral model of an Alpha processor. It executes all instructions in a single cycle and keeps track of internal state such as registers, status flags, TLBs, caches, and memory. We use the Reference model to check the execution of our Alpha Core implementation. As each instruction retires in our Verilog model, we compare the internal state of the two models and immediately flag any discrepancies.

The design of this simulator allowed us to leverage self-checking, pseudo-random tests to help reduce testing overhead. As we will discuss in the next section on our test plan, we also followed this strategy in our testing of the memory hierarchy.

3.3 Test Plan and Current Status

In planning to efficiently manage verification overhead, the key strategies of our test plan were to focus on testing the fully integrated memory hierarchy and to heavily leverage self-checking, pseudo-random tests in our verification effort.

Memory Hierarchy Testing. The verification of the memory hierarchy in particular depended on pseudo-random tests. Our test plan split this verification task into three distinct phases: initial focused tests, continuous pseudo-random stress tests, and final coverage-driven focused tests. The initial focused tests were a series of simple tests lasting in total 1-2 weeks and intended to debug the basic operations of each module. The continuous pseudo-random stress tests were a set of self-checking tests that performed memory accesses to a small range of memory blocks, triggering a large amount of false sharing communication throughout the system. We intended this phase to constitute the majority of our verification time, anywhere from 8-12 months. The duration of the third phase of coverage-driven focused tests depended on the effectiveness of the pseudo-random test phase.

By the time the project ended, the system had been in pseudo-random stress tests for roughly five months. The stress test was running continuously on a cluster of approximately 23 machines (13 server-class machines and 10 desktop machines) with a total of 51 processors. The tests exercised approximately 80% of our design and our bug rate was relatively low at less than ten per month. From this information, we estimated that the system verification was on schedule and had a few more months of stress test before finishing this phase. Verification of the Alpha Core however had been proceeding even faster and was nearing completion.

Alpha Core Testing. The Alpha Core, as described above, heavily leveraged the efforts of the Alpha 21364 verification team. We used their focused test stimuli directly, and these tests accounted for approximately one third of the Alpha Core test plan. The remaining two thirds of the test plan was based on pseudo-random tests driven by the STIM-P tool. The goal of these pseudo-random tests was to exercise the various instructions and trap conditions such that we could maximize the coverage of a set of designer-specified coverage points. These coverage points were chosen to indicate when complicated aspects of the design had been tested. (Due to the Alpha Core's pipeline design, line coverage is very easy to achieve, and so not a good measurement of testing effectiveness. During the early test phases, almost 95% of the lines were covered by tests.)

Unlike in the memory hierarchy test plan, the focused and pseudo-random test phases in the Alpha Core test plan were run concurrently. Basically, as a particular focused test was finished, the instructions under test were added to the pseudo-random stimuli. When the project ended, we had completed 98% of the focused tests and had been running pseudo-random tests for 6 months. Our tests had exercised approximately 75% of the designer-specified coverage points. Based on this information, we estimated that two more months were required to complete the Alpha Core verification.

4. Physical Design

An ASIC design process offers considerable advantages in managing physical design complexity, however the advantages do have some cost. The major advantage of the approach is that software tools can be used to automate much of the low level gate layout, and the resulting chip layout can be easily transferred to an ASIC vendor for fabrication. The trade-off is that the design is typically less efficient, in speed and power, than a comparable design achieved by manual gate layout.

In our case, the advantages of ASIC design greatly outweighed the disadvantages. We chose to use IBM and their Cu-11 [7] process. IBM's ASIC foundry validates its chip implementations through a very thorough testing methodology. As part of this testing methodology, IBM forbids the use of tri-state busses and also only allows "safe" one-hot multiplexors. The restrictions are in place because if the control logic is faulty, the output of these physical structures is unpredictable and so physical validation may not be possible. Unfortunately, tri-state busses and optimized one-hot multiplexors are very useful in high performance circuits, as they can greatly simplify and speed up some logic. We were eventually able to use the faster one-hot multiplexors, but not tri-state busses. This latter issue complicated the design of the Intra-Chip Switch by forcing more levels of logic leading into the switch.

A variety of early design decisions had to be re-examined given the constraints of the memory cells available in the ASIC library. For example, we had to reduce our target clock speed from 500MHz to 400MHz, given the SRAM and register array access speeds. For the same reason, we also reduced our L1 cache from a 64K, 2-way associative design to a 32K, direct-mapped design. This design change affected not only the access times in L1, but also the duplicate L1 tag lookups in the L2.

Apart from the SRAMs and register arrays, our design also had several fully-associative memory structures which, due to constraints of the ASIC methodology, were limited to roughly 16 entries. In the case of our processor's branch target buffer (BTB), however, the necessary timing limited us to only eight entries. Since our simulation showed that a BTB of less than 16 entries would have little effect on performance, we decided to drop the BTB and simplify the physical design.

The largest impact of the limited size fully-associative memory structures, however, was on the Translation Lookaside Buffer (TLB). Not surprisingly, a 16-entry fully-associative TLB was not a valid option due to performance concerns. We believed our only option was to build a four-way associative 256-entry TLB, implemented as four separate 64-entry register arrays that could be checked in parallel. This design also reduced the level to which we could support Alpha granularity hints [14], which are used to allow multiple page sizes to be used within an address space. Fortunately, our simulations showed this TLB design did not significantly impact our OLTP performance.

Module	Size mm^2
Alpha Core	2.1 (0.3)
L1 Cache	2.0 (5.8)
L2 Cache	2.3 (9.9)
Coherence Engines	1.4 (1.2)
I/O Queues	1.1 (0.6)
Router	0.5 (1.1)
Intra-Chip Switch	1.4 (0.0)
Memory Controller	4.8 (0.0)
System Controller	1.0 (0.0)
Miscellaneous	54.6 (1.0)
Total CMP	282.6 (133.13)

Table 1: Estimated module sizes for the processing node. Estimated memory sizes are included in parentheses.

In addition to the memory cells, we also used a PCI bus interface[1] and an SDRAM interface core from IBM's library. These two cores handled the chip's interface logic to the PCI bus and SDRAM bus, respectively. We were able to use the PCI core directly, however the SDRAM core was wrapped by a generic memory interface that hid the ECC calculation. As we needed to perform our own ECC calculation in order to store the system-level coherence directory, we obtained access to the low-level SDRAM interface from IBM.

Design Status. All modules, except for the System Controller and Intra-Chip Switch, reached functioning Verilog stage, and were undergoing timing re-designs at the time the project was concluded. The machine translation of the memory hierarchy modules produced seemingly high quality Verilog code. When this code was synthesized, the Router, Input and Output Queues were all within 1ns of the target 400MHz timing. The L1, L2, and Coherence Engines had undergone the first pass for timing re-designs and the long paths of each module were within 2.5ns of timing.

Table 1 shows the estimated size of the modules in the

[1] The PCI bus interface core handled the mechanics of the connecting to the PCI bus. The PCI interface module in Figure 3 implemented the interface between the PCI subsystem and the memory hierarchy.

processing node, along with the size of the node itself. The Miscellaneous entry includes items such as clock and boundary logic. The estimated maximum power dissipation of a processing node was 44W at 400MHz and V_{dd}=1.5V. The power estimates for some basic Cu-11 structures were not available, so we based our estimates on values from the earlier SA-27E [8] process.

5. Conclusions

The trend toward increasing processor complexity, driven by the quest for further exploitation of instruction level parallelism, has produced diminishing returns for important server-class workloads, such as database applications and web servers. These workloads simply do not exhibit the potential for ILP that exists in scientific and engineering applications. Regardless, this increased complexity has greatly lengthened all phases of chip design, particularly verification and physical design, in addition to inflating team sizes and budgets to nearly impractical levels across the industry.

In Piranha, we sought to build a processor that would address both commercial workload performance and project complexity. The Piranha architecture, based on chip multiprocessing, provides an excellent platform for commercial workloads given the abundance of thread level parallelism available. In simulation, the Piranha architecture, built in a conservative ASIC process, can provide more than a 2x performance improvement over a conventional, next-generation design. Furthermore, this performance advantage is achieved with significant reductions in design and verification complexity. Piranha's CMP architecture fundamentally emphasizes replication over monolithic complexity. Furthermore, given our workload focus, we were able to make additional architectural trade-offs, such as using single-issue, in-order processor cores, that reduced project complexity without sacrificing overall performance.

We furthered complemented the architectural decisions with specific measures to reduce the effort incurred in the mechanics of the verification and physical design phases. Our most important measure was to employ a novel C++-based ASIC design flow that enabled both a very high performance logic simulator and the use of industry-standard physical design tools. The high performance logic simulator allowed us to greatly reduce verification overhead, as it enabled a very efficient testing of the integrated design, while the ASIC design flow, with its use of industry-standard tools, automated much of the low level physical design layout and provided a clear path to fabrication.

Together, these measures helped us design a high performance server-class processor project with a level of complexity appropriate for a group much smaller than that of typical processor projects. Working for slightly more than a year, a group of less than 20 people brought the Piranha design to the cusp of a physical prototype. During this period, the chip design was specified as an RTL model, the logic design was substantially verified, and the physical design was undergoing the necessary work to reach an aggressive operating frequency of 400MHz using an ASIC process.

Acknowledgments

The Piranha project was a joint effort between Compaq’s Research Division and the Non-Stop Hardware Division. At the time the project concluded, the team members were Mark Dorland, Ray Harlan, Tom Heynemann, Sachin Idgunji, Scott Johnson, Dan Joyce, David Lowell, Uma Maheswari, Joel McCormack, Harland Maxwell, Harold Miller, Scott Smith, Jeff Sprouse, Hemendra Talesara, Alex Wakefield, Robert Willey, Yuan Yu, and the authors. Former team member Andreas Nowatzyk contributed significantly to the Piranha architecture and methodology.

References

1 P. Bannon. Alpha 21364: A Scalable Single-chip SMP. Presented at the *Microprocessor Forum ‘98 (http://www.digital.com/alphaoem/microprocessorforum.htm)*, October 1998.

2 L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *27th International Symposium on Computer Architecture*, pages 282-293, June, 2000.

3 L. A. Barroso, K. Gharachorloo, A. Nowatzyk, and B. Verghese. Impact of Chip-Level Integration on Performance of OLTP Workloads. In *6th International Symposium on High-Performance Computer Architecture*, pages 3-14, January 2000.

4 L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory System Characterization of Commercial Workloads. In *25th Annual International Symposium on Computer Architecture*, pages 3-14, June 1998.

5 CLevel Design, http://www.cleveldesign.com/home.html, July 31, 2000.

6 S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous Multithreading: A Platform for Next-Generation Processors. In *IEEE Micro*, pages 12–19, October 1997.

7 IBM Microelectronics. ASIC Cu-11 Databook. International Business Machines, 2001.

8 IBM Microelectronics. ASIC SA27E Databook. International Business Machines, 1999.

9 D. Joyce, A. Nowatzyk, R. Stets. A C++ ASIC Design Methodology Facilitated by a C++-Verilog Translator. In *10th International HDL Conference*, pages 180-187, February, 2001.

10 K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker. Performance Characterization of the Quad Pentium Pro SMP Using OLTP Workloads. In *25th Annual International Symposium on Computer Architectur*e, pages 15-26, June 1998.

11 J. Lo, L. A. Barroso, S. Eggers, K. Gharachorloo, H. Levy, and S. Parekh. An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors. In *25th Annual International Symposium on Computer Architecture*, June 1998.

12 P. Ranganathan, K. Gharachorloo, S. Adve, and L. A. Barroso. Performance of Database Workloads on Shared- Memory Systems with Out-of-Order Processors. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 307-318, October 1998.

13 M. Rosenblum, E. Bugnion, S. Herrod, and S. Devine. Using the SimOS Machine Simulator to Study Complex Computer Systems. In *ACM Transactions on Modeling and Computer Simulation*, Vol. 7, No. 1, pages 78-103, January 1997.

14 R. L. Sites and R. T. Witek. Alpha AXP Architecture Reference Manual (second edition). Digital Press, 1995.

15 Standard Performance Council. The SPEC95 CPU Benchmark Suite. *http://www.specbench.org*, 1995.

16 Transaction Processing Performance Council. TPC Benchmark B Standard Specification Revision 2.0. June 1994