In Proceedings of the 27th Annual International Symposium on Computer Architecture, June 2000

Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing

Luiz André Barroso, Kourosh Gharachorloo, Robert McNamara†, Andreas Nowatzyk, Shaz Qadeer†, Barton Sano, Scott Smith‡, Robert Stets, and Ben Verghese

Western Research Laboratory
Compaq Computer Corporation
Palo Alto, CA 94301

†Systems Research Center
Compaq Computer Corporation
Palo Alto, CA 94301

‡NonStop Hardware Development
Compaq Computer Corporation
Austin, TX 78728

Abstract

The microprocessor industry is currently struggling with higher development costs and longer design times that arise from exceedingly complex processors that are pushing the limits of instruction-level parallelism. Meanwhile, such designs are especially ill suited for important commercial applications, such as on-line transaction processing (OLTP), which suffer from large memory stall times and exhibit little instruction-level parallelism. Given that commercial applications constitute by far the most important market for high-performance servers, the above trends emphasize the need to consider alternative processor designs that specifically target such workloads. The abundance of explicit thread-level parallelism in commercial workloads, along with advances in semiconductor integration density, identify chip multiprocessing (CMP) as potentially the most promising approach for designing processors targeted at commercial servers.

This paper describes the *Piranha* system, a research prototype being developed at Compaq that aggressively exploits chip multiprocessing by integrating eight simple Alpha processor cores along with a two-level cache hierarchy onto a single chip. Piranha also integrates further on-chip functionality to allow for scalable multiprocessor configurations to be built in a glueless and modular fashion. The use of simple processor cores combined with an industry-standard ASIC design methodology allow us to complete our prototype within a short time-frame, with a team size and investment that are an order of magnitude smaller than that of a commercial microprocessor. Our detailed simulation results show that while each Piranha processor core is substantially slower than an aggressive next-generation processor, the integration of eight cores onto a single chip allows Piranha to outperform next-generation processors by up to 2.9 times (on a per chip basis) on important workloads such as OLTP. This performance advantage can approach a factor of five by using full-custom instead of ASIC logic. In addition to exploiting chip multiprocessing, the Piranha prototype incorporates several other unique design choices including a shared second-level cache with no inclusion, a highly optimized cache coherence protocol, and a novel I/O architecture.

1 Introduction

High-end microprocessor designs have become increasingly more complex during the past decade, with designers continuously pushing the limits of instruction-level parallelism and speculative out-of-order execution. While this trend has led to significant performance gains on target applications such as the SPEC benchmark [40], continuing along this path is becoming less viable due to substantial increases in development team sizes and design times [18]. Furthermore, more complex designs are yielding diminishing returns in performance even for applications such as SPEC.

Meanwhile, commercial workloads such as databases and Web applications have surpassed technical workloads to become the largest and fastest-growing market segment for high-performance servers. A number of recent studies have underscored the radically different behavior of commercial workloads such as on-line transaction processing (OLTP) relative to technical workloads [4,7, 8,21,28,34,36]. First, commercial workloads often lead to inefficient executions dominated by a large memory stall component. This behavior arises from large instruction and data footprints and high communication miss rates which are characteristic for such workloads [4]. Second, multiple instruction issue and out-of-order execution provide only small gains for workloads such as OLTP due to the data-dependent nature of the computation and the lack of instruction-level parallelism [35]. Third, commercial workloads do not have any use for the high-performance floating-point and multimedia functionality that is implemented in modern microprocessors. Therefore, it is not uncommon for a high-end microprocessor to be stalling most of the time while executing commercial workloads, leading to a severe under-utilization of its parallel functional units and high-bandwidth memory system. Overall, the above trends further question the wisdom of pushing for more complex processor designs with wider issue and more speculative execution, especially if the server market is the target.

Fortunately, increasing chip densities and transistor counts provide architects with several alternatives for better tackling design complexities in general, and the needs of commercial workloads in particular. For example, the next-generation Alpha 21364 aggressively exploits semiconductor technology trends by including a scaled 1GHz 21264 core (i.e., shrink of the current Alpha processor core to 0.18um technology), two levels of caches, memory controller, coherence hardware, and network router all on a single die [2]. The tight coupling of these modules enables a more efficient and lower latency memory hierarchy which can substantially improve the performance of commercial workloads [3]. Furthermore, the reuse of an existing high-performance processor core in designs such as the Alpha 21364 effectively addresses the design complexity issues and provides better time-to-market without sacrificing server performance.

Higher transistor counts can also be used to exploit the inherent and explicit thread-level (or process-level) parallelism which is abundantly available in commercial workloads to better utilize on-chip resources. Such parallelism typically arises from relatively independent transactions or queries initiated by different clients, and has traditionally been used to hide I/O latency in such workloads. Previous studies have shown that techniques such as simultaneous multithreading (SMT) can provide a substantial

performance boost for database workloads [27]. In fact, the Alpha 21464 (successor to Alpha 21364) is planning to combine aggressive chip-level integration (see previous paragraph) along with an eight-instruction-wide out-of-order processor with SMT support for four simultaneous threads [13]. An alternative approach, often referred to as chip multiprocessing (CMP) [15], involves integrating multiple (possibly simpler) processor cores onto a single chip. This approach has been adopted by the next-generation IBM Power4 design which integrates two superscalar cores along with a shared second-level cache [9]. While the SMT approach is superior in single-thread performance (important for workloads without explicit thread-level parallelism), it is best suited for very wide-issue processors which are more complex to design. In comparison, CMP advocates using simpler processor cores at a potential loss in single-thread performance, but compensates in overall throughput by integrating multiple such cores. Furthermore, CMP naturally lends itself to a hierarchically partitioned design with replicated modules, allowing chip designers to use short wires as opposed to costly and slow long wires that can adversely affect cycle time.

This paper presents a detailed description and evaluation of Piranha, a research prototype being developed at Compaq (jointly by Corporate Research and NonStop Hardware Development) to explore chip multiprocessing architectures targeted at parallel commercial workloads.[1] The centerpiece of the Piranha architecture is a highly-integrated *processing node* with eight simple Alpha processor cores, separate instruction and data caches for each core, a shared second-level cache, eight memory controllers, two coherence protocol engines, and a network router all on a single die. Multiple such processing nodes can be used to build a glueless multiprocessor in a modular and scalable fashion.

The primary goal of the Piranha project is to build a system that achieves superior performance on commercial workloads (especially OLTP) with a small team, modest investment, and a short design time. These requirements heavily influence the design decisions and methodology used in implementing our prototype. First, we have opted for extremely simple processor cores using a single-issue in-order eight-stage pipelined design. Second, we have opted for a semi-custom design based on industry-standard ASIC methodologies and tools, making heavy use of synthesis with standard cells. To achieve acceptable performance, we rely on a state-of-the-art 0.18um ASIC process and make limited use of custom-designed memory cells for a few time- or area-critical memory structures. Nonetheless, some of our modules are larger in area and our target clock speed is about half of what could be achieved with custom logic in the same process technology.

We present a detailed performance evaluation of the Piranha design based on full system simulations, including operating system activity, of the Oracle commercial database engine running under Compaq Tru64 Unix. Our results show that although each Piranha processor core is substantially slower than an aggressive next-generation processor, the integration of eight cores onto a single chip allows Piranha to outperform next-generation processors by about 2.3 to 2.9 times on a per chip basis on important commercial workloads. The true potential of the Piranha architecture is more fairly judged by considering a full-custom design. This approach clearly requires a larger design team and investment, but still maintains the relatively low complexity and short design time characteristics. Our results show that a more custom design can enhance Piranha's performance advantage on workloads such as OLTP to about 5.0 times better (on a per chip basis) relative to next-generation processors. These results clearly indicate that focused designs such as Piranha that directly target commercial server applications can substantially outperform general-purpose microprocessor designs with much higher complexity.

1. The project name is motivated by the analogy of seemingly small fish that in concert can vanquish a much larger creature.

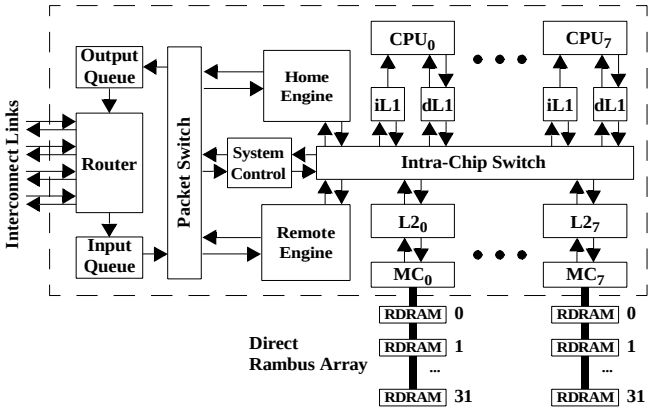


Figure 1. Block diagram of a single-chip Piranha processing node.

In addition to exploring chip multiprocessing, the Piranha architecture incorporates a number of other novel ideas. First, the design of the shared second-level cache uses a sophisticated protocol that does not enforce inclusion in first-level instruction and data caches in order to maximize the utilization of on-chip caches. Second, the cache coherence protocol among nodes incorporates a number of unique features that result in fewer protocol messages and lower protocol engine occupancies compared to previous protocol designs. Finally, Piranha has a unique I/O architecture, with an *I/O node* that is a full-fledged member of the interconnect and the global shared-memory coherence protocol.

The rest of the paper is structured as follows. Section 2 presents an overview of the Piranha architecture. We next describe our experimental methodology, including the workloads and other architectures that we study. Section 4 presents detailed performance results for the Piranha prototype and considers more custom implementations that better illustrate the full potential of our approach. Our design methodology and implementation status are described in Section 5. Finally, we discuss related work and conclude.

2 Piranha Architecture Overview

Figure 1 shows the block diagram of a single Piranha processing chip. Each *Alpha CPU core (CPU)* is directly connected to dedicated instruction (*iL1*) and data cache (*dL1*) modules. These first-level caches interface to other modules through the *Intra-Chip Switch (ICS)*. On the other side of the ICS is a logically shared *second level cache (L2)* that is interleaved into eight separate modules, each with its own controller, on-chip tag, and data storage. Attached to each L2 module is a *memory controller (MC)* which directly interfaces to one bank of up to 32 direct Rambus DRAM chips. Each memory bank provides a bandwidth of 1.6GB/sec, leading to an aggregate bandwidth of 12.8 GB/sec. Also connected to the ICS are two protocol engines, the *Home Engine (HE)* and the *Remote Engine (RE)*, which support shared memory across multiple Piranha chips. The interconnect subsystem that links multiple Piranha chips consists of a *Router (RT)*, an *Input Queue (IQ)*, an *Output Queue (OQ)* and a *Packet Switch (PS)*. The total interconnect bandwidth (in/out) for each Piranha processing chip is 32 GB/sec. Finally, the *System Control (SC)* module takes care of miscellaneous maintenance-related functions (e.g., system configuration, initialization, interrupt distribution, exception handling, performance monitoring). It should be noted that the various modules communicate exclusively through the connections shown in Figure 1, which also represent

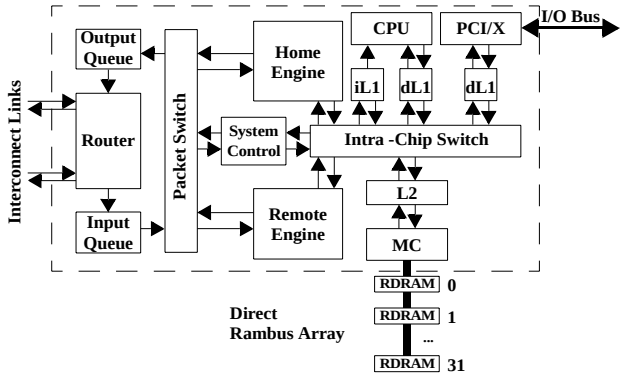


Figure 2. Block diagram of a single-chip Piranha I/O node.

the actual signal connections. This modular approach leads to a strict hierarchical decomposition of the Piranha chip which allows for the development of each module in relative isolation along with well defined transactional interfaces and clock domains.

While the Piranha processing chip is a complete multiprocessor system on a chip, it does not have any I/O capability. The actual I/O is performed by the Piranha I/O chip, shown in Figure 2, which is relatively small in area compared to the processing chip. Each I/O chip is a stripped-down version of the Piranha processing chip with only one CPU and one L2/MC module. The router on the I/O chip is also simplified to support only two instead of four links, thus alleviating the need for a routing table. From the programmer's point of view, the CPU on the I/O chip is indistinguishable from one on the processing chip. Similarly, the memory on the I/O chip fully participates in the global cache coherence scheme. The presence of a processor core on the I/O chip provides several benefits: it enables optimizations such as scheduling device drivers on this processor for lower latency access to I/O, or it can be used to virtualize the interface to various I/O devices (e.g., by having the Alpha core interpret accesses to virtual control registers).

Except for the PCI/X interface, which is available in our ASIC library, most of the modules on the I/O chip are identical in design to those on the processing chip. To simplify the design, we reuse our first-level data cache module (dL1) to interface to the PCI/X module. The dL1 module also provides the PCI/X with address translation, access to I/O space registers, and interrupt generation. The Piranha I/O chip may also be customized to support other I/O standards such as Fiber Channel and System I/O.

Figure 3 shows an example configuration of a Piranha system with both processing and I/O chips. The Piranha design allows for glueless scaling up to 1024 nodes, with an arbitrary ratio of I/O to processing nodes (which can be adjusted for a particular workload). Furthermore, the Piranha router supports arbitrary network topologies and allows for dynamic reconfigurability. One of the underlying design decisions in Piranha is to treat I/O in a uniform manner as a full-fledged member of the interconnect. In part, this decision is based on the observation that available inter-chip bandwidth is best invested in a single switching fabric that forms a global resource which can be dynamically utilized for both memory and I/O traffic.

One of the key design decisions in Piranha was to remain binary compatible with the Alpha software base, including both applications and system software (e.g., compilers, operating system, etc.). Therefore, user applications will run without any modification, and we are expecting a minimal porting effort for the OS (Tru64 Unix).

The remaining sections provide more detail about the various modules in the Piranha architecture.

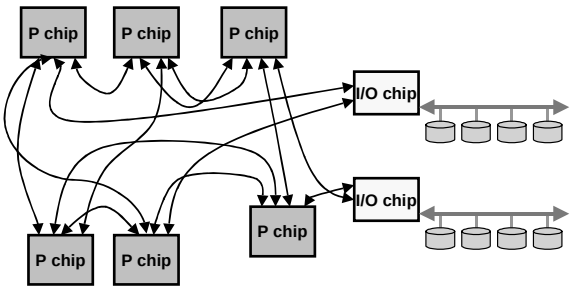


Figure 3. Example configuration for a Piranha system with six processing (8 CPUs each) and two I/O chips.

2.1 Alpha CPU Core and First-Level Caches

The processor core uses a single-issue, in-order design capable of executing the Alpha instruction set [39]. It consists of a 500MHz pipelined datapath with hardware support for floating-point operations. The pipeline has 8 stages: instruction fetch, register-read, ALU 1 through 5, and write-back. The 5-stage ALU supports pipelined floating-point and multiply instructions. However, most instructions execute in a single cycle. The processor core includes several performance enhancing features including a branch target buffer, pre-compute logic for branch conditions, and a fully bypassed datapath. The processor core interfaces to separate first-level instruction and data caches designed for single-cycle latency. We use 64KB two-way set-associative, blocking caches with virtual indices and physical tags. The L1 cache modules include tag compare logic, instruction and data TLBs (256 entries, 4-way associative), and a store buffer (data cache only). We also maintain a 2-bit state field per cache line, corresponding to the four states in a typical MESI protocol. For simplicity, the instruction and data caches use virtually the same design. Therefore, unlike other Alpha implementations, the instruction cache is kept coherent by hardware. Treating the instruction and data caches in the same way also simplifies our no-inclusion policy at the L2 level.

2.2 Intra-Chip Switch

Conceptually, the intra-chip switch (ICS) is a crossbar that interconnects most of the modules on a Piranha chip. However, managing the data transfers from its 27 clients efficiently poses a number of implementation challenges, such as arbitration, flow control, and layout. The ICS is also the primary facility for decomposing the Piranha design into relatively independent, isolated modules. In particular, the transactional nature of the ICS allows us to add or remove pipeline stages during the design of various modules without compromising the overall Piranha timing.

The ICS uses an uni-directional, push-only interface. The initiator of a transaction always sources data. If the destination of a transaction is ready, the ICS schedules the data transfer according to datapath availability. A grant is issued to the initiator to commence the data transfer at a rate of one 64-bit word per cycle without any further flow control. Concurrently, the destination receives a request signal that identifies the initiator and the type of transfer. Transfers are atomic, and the implied ordering properties are exploited in supporting intra-chip coherence.

Each port to the ICS consists of two independent 64-bit datapaths (plus 8-bit parity/ECC bits) for sending and receiving data. The ICS supports back-to-back transfers without dead-cycles between transfers. In order to reduce latency, modules are allowed to issue the target destination of a future request ahead of the actual transfer request. This hint is used by the ICS to pre-allocate datapaths and to speculatively assert the requester's grant signal.

The ICS is implemented by using a set of eight internal datapaths that run along the center of the Piranha chip. Given that the internal ICS capacity is 32 GB/sec or about 3 times the available memory bandwidth, achieving an optimal schedule is not critical to achieve good performance.

The ICS supports two logical lanes (low- and high-priority) that are used to avoid intra-chip cache coherence protocol deadlocks. Instead of adding extra datapaths, multiple lanes are supported by two ready lines with distinct IDs for each module. An initiator can specify the appropriate lane for a transaction by using the corresponding ID for the destination.

2.3 Second-Level Cache

Piranha's second-level cache (L2) is a 1MB unified instruction/data cache which is physically partitioned into eight banks and is logically shared among all CPUs. The L2 banks are interleaved using the lower address bits of a cache line's physical address (64-byte line). Each bank is 8-way set-associative and uses a round-robin (or *least-recently-loaded*) replacement policy if an invalid block is not available. Each bank has its own control logic, an interface to its private memory controller, and an ICS interface used to communicate with other chip modules. The L2 controllers are responsible for maintaining intra-chip coherence, and cooperate with the protocol engines to enforce inter-chip coherence.

Since Piranha's aggregate L1 capacity is 1MB, maintaining data inclusion in our 1MB L2 can potentially waste its full capacity with duplicate data. Therefore, Piranha opts for not maintaining the inclusion property. Although non-inclusive on-chip cache hierarchies have been previously studied in the context of a single-CPU chip [20], the use of this technique in the context of a CMP leads to interesting issues related to coherence and allocation/replacement policies. To simplify intra-chip coherence and avoid the use of snooping at L1 caches, we keep a duplicate copy of the L1 tags and state at the L2 controllers. Each controller maintains tag/state information for L1 lines that map to it given the address interleaving. The total overhead for the duplicate L1 tag/state across all controllers is less than 1/32 of the total on-chip memory.

In order to lower miss latency and best utilize the L2 capacity, L1 misses that also miss in the L2 are filled directly from memory without allocating a line in the L2. The L2 effectively behaves as a very large victim cache that is filled only when data is replaced from the L1s. Hence, even clean lines that are replaced from an L1 may cause a write-back to the L2. To avoid unnecessary write-backs when multiple L1s have copies of the same line, the duplicate L1 state is extended to include the notion of *ownership*. The owner of a line is either the L2 (when it has a valid copy), an L1 in the exclusive state, or one of the L1s (typically the last requester) when there are multiple sharers. Based on this information, the L2 makes the decision of whether an L1 should write back its data and piggybacks this information with the reply to the L1's request (that caused the replacement). In the case of multiple sharers, a write-back happens only when an owner L1 replaces the data. The above approach provides a near-optimal replacement policy without affecting our L2 hit time. We ruled out alternative solutions that require checking all L1 states or the state of the victim in the L2 since they would require multiple tag lookup cycles in the critical path of an L2 hit.

Intra-chip coherence protocol. The L2 controllers are responsible for enforcing coherence within a chip. Each controller has complete and exact information about the on-chip cached copies for the subset of lines that map to it. On every L2 access, the duplicate L1 tag/state and the tag/state of the L2 itself are checked in parallel. Therefore, our intra-chip coherence has similarities to a full-map centralized directory-based protocol. Information about sharing of data across chips is kept in the directory, which is stored in DRAM and accessed through the memory controller (see Section 2.5.2). Full interpretation and manipulation of the directory bits is only done by the protocol engines. However, the L2 controllers can partially interpret the directory information to determine whether a line is cached by a remote node(s) and if so, whether it is cached exclusively. This partial information, which is kept in the L2 and duplicate L1 states, allows the L2 controller at home to avoid communicating with the protocol engines for the majority of local L1 requests. In many cases this partial information also avoids having to fetch the directory from memory when a copy of the line is already cached in the chip.

A memory request from an L1 is sent to the appropriate L2 bank based on the address interleaving. Depending on the state at the L2, the L2 can possibly (a) service the request directly, (b) forward the request to a local (owner) L1, (c) forward the request to one of the protocol engines, or (d) obtain the data from memory through the memory controller (only if the home is local). The L2 is also responsible for all on-chip invalidations, whether triggered by local or remote requests. The ordering characteristics of the intra-chip switch allow us to eliminate the need for acknowledgments for on-chip invalidations. Invalidating and forwarding requests to remote nodes are handled through the protocol engines. Requests forwarded to the home engine carry a copy of the directory, which is updated by the home engine and later written back to memory. In all forwarding cases, the L2 keeps a *request pending* entry which is used to block conflicting requests for the duration of the original transaction. A small number of such entries are supported at each L2 controller in order to allow concurrent outstanding transactions.

2.4 Memory Controller

Piranha has a high bandwidth, low latency memory system based on direct Rambus RDRAM. In keeping with our modular design philosophy, there is one memory controller and associated RDRAM channel for each L2 bank, for a total of eight memory controllers. Each Rambus channel can support up to 32 RDRAM chips. In the 64Mbit memory chip generation, each Piranha processing chip can support a total of 2GB of physical memory (8GB/32GB with 256Mb/1Gb chips). Each RDRAM channel has a maximum data rate of 1.6GB/sec, providing a maximum local memory bandwidth of 12.8GB/sec per processing chip. The latency for a random access to memory over the RDRAM channel is 60ns for the critical word, and an additional 30ns for the rest of the cache line.

Unlike other chip modules, the memory controller does not have direct access to the intra-chip switch. Access to memory is controlled by and routed through the corresponding L2 controller. The L2 can issue read/write requests to memory, at the granularity of a cache line, for both data and the associated directory.

The design of the memory controller consists of two parts: the Rambus Access Controller (RAC) and the memory controller engine. The RAC is provided by Rambus and incorporates all the high-speed interface circuitry. The memory controller engine functionality includes the MC/L2 interface and the scheduling of memory accesses. Most of the complexity comes from deciding what pages to keep open across the various devices. In a fully populated Piranha chip, we have as many as 2K (512-byte) pages open. A hit to an open page reduces the access latency from 60ns to 40ns. Our simulations show that keeping pages open for about 1 microsecond will yield a hit rate of over 50% on workloads such as OLTP.

2.5 Protocol Engines

As shown in Figure 1, the Piranha processing node has two separate protocol engines that are used to support shared-memory across multiple nodes. The *home engine* is responsible for exporting memory whose home is at the local node, while the *remote*

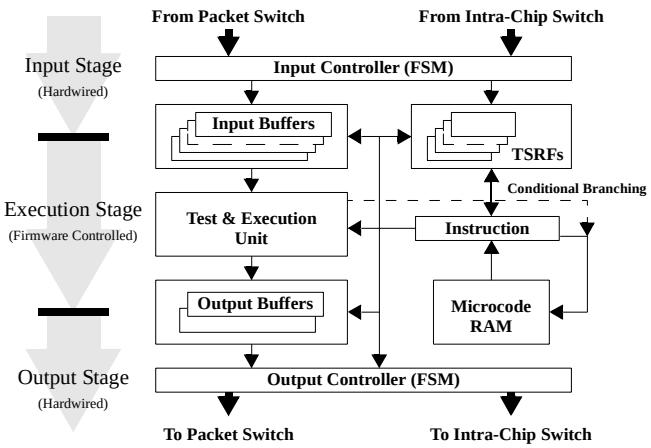


Figure 4. Block diagram of a protocol engine.

engine imports memory whose home is remote. The following sections describe the protocol engine design, the directory storage, and the inter-node coherence protocol in more detail.

2.5.1 Protocol Engine Structure

The protocol engines in Piranha are implemented as microprogrammable controllers, with the home and remote engines being virtually identical except for the microcode that they execute. Our approach uses the same design philosophy as the protocol engines used in the S3.mp project [32]. Figure 4 shows a high-level block diagram of one protocol engine consisting of three independent (and decoupled) stages: the input controller, the microcode-controlled execution unit, and the output controller. The input controller receives messages from either the local node or the external interconnect, while the output controller sends messages to internal or external destinations.

The bottom right section of Figure 4 depicts the micro-sequencer which consists of a microcode memory and a current instruction register. The microcode memory supports 1024 21-bit-wide instructions (the current protocol uses about 500 microcode instructions per engine). Each microcode instruction consists of a 3-bit opcode, two 4-bit arguments, and a 10-bit address that points to the next instruction to be executed. Our design uses the following seven instruction types: SEND, RECEIVE, LSEND (to local node), LRECEIVE (from local node), TEST, SET, and MOVE. The RECEIVE, LRECEIVE, and TEST instructions behave as multi-way conditional branches that can have up to 16 different successor instructions, achieved by OR-ing a 4-bit condition code into the least significant bits of the 10-bit next-instruction address field. To allow for 500MHz operation, we use an interleaved execution model whereby we fetch the next instruction for an even-addressed (odd-addressed) thread while executing the instruction for an odd-addressed (even-addressed) thread.

The actual protocol code is specified at a slightly higher level with symbolic arguments, and C-style code blocks, and a sophisticated microcode assembler is used to do the appropriate translation and mapping to the microcode memory. Typical cache coherence transactions require only a few instructions at each engine that handles the transaction. For example, a typical read transaction to a remote home involves a total of four instruction at the remote engine of the requesting node: a SEND of the request to the home, a RECEIVE of the reply, a TEST of a state variable, and an LSEND that replies to the waiting processor at that node.

On a new transaction, the protocol engine allocates an entry from the transaction state register file (TSRF) that represents the state of this thread (e.g., addresses, program counter, timer, state variables, etc.). A thread that is waiting for a response from a local or remote node has its TSRF entry set to a waiting state, and the incoming response is later matched with this entry based on the transaction address. Our design supports a total of 16 TSRF entries per protocol engine to allow for concurrent protocol transactions.

We believe our design provides a nice balance between flexibility (e.g., for late binding of protocol) and performance. While the design is less flexible than using a general-purpose processor as in FLASH [24], the specialized (more powerful) instructions lead to much lower protocol engine latency and occupancy.

2.5.2 Directory Storage

The Piranha design supports directory data with virtually no memory space overhead by computing ECC at a coarser granularity and utilizing the unused bits for storing the directory information [31,38]. ECC is computed across 256-bit boundaries (typical is 64-bit), leaving us with 44 bits for directory storage per 64-byte line. Compared to having a dedicated external storage and datapath for directories, this approach leads to lower cost by requiring fewer components and pins, and provides simpler system scaling. In addition, we leverage the low latency, high bandwidth path provided by the integration of memory controllers on the chip.

We use two different directory representations depending on the number of sharers: limited pointer [1] and coarse vector [14]. Two bits of the directory are used for state, with 42 bits available for encoding sharers. The directory is not used to maintain information about sharers at the home node. Furthermore, directory information is maintained at the granularity of a node (not individual processors). Given a 1K node system, we switch to coarse vector representation past 4 remote sharing nodes.

2.5.3 Inter-node Coherence Protocol

Piranha uses an invalidation-based directory protocol with support for four request types: *read*, *read-exclusive*, *exclusive* (requesting processor already has a shared copy), and *exclusive-without-data*[2]. We also support the following features: clean-exclusive optimization (an exclusive copy is returned to a read if there are no other sharers), reply forwarding from remote owner, and eager exclusive replies (ownership given before all invalidations are complete). Invalidation acknowledgments are gathered at the requesting node. Finally, the protocol does not depend on point-to-point order, thus allowing the external interconnect to use techniques such as adaptive routing.

A unique property of our protocol is that it avoids the use of negative acknowledgment (NAK) messages and the corresponding retries. There are two reasons why NAKs are used in scalable coherence protocols: (i) requests are NAKed to avoid deadlock when outgoing network lanes back up, and (ii) requests are NAKed due to protocol races where a request fails to find the data at the node it is forwarded to. We avoid the *first* use of NAKs by using three virtual lanes (I/O, L, H). Explaining the need for the I/O lane is beyond the scope of this paper. The low priority lane (L) is used by requests sent to a home node (except for write-back/replacement requests which use H), while the high priority lane (H) is used by forwarded requests and all replies. Our deadlock solution also relies on sufficient buffering in the network (explained later). We avoid the *second* use of NAKs by guaranteeing that forwarded requests can always be serviced by their target nodes. For example, when an owner node writes back its data to home, it maintains a valid copy of the data until the home

2. This corresponds to the Alpha write-hint instruction (wh64) which indicates that the processor will write the entire cache line, thus avoiding a fetch of the line's current contents (e.g., useful in copy routines).

acknowledges the writeback (allowing it to satisfy forwarded requests). There are also cases where a forwarded request may arrive at an owner node too early, i.e., before the owner node has received its own data. In this case, we delay the forwarded request until the data is available.[3]

The lack of NAKs/retries leads to a more efficient protocol and provides several important and desirable characteristics. First, since an owner node is guaranteed to service a forwarded request, the protocol can complete all directory state changes immediately. This property eliminates the need for extra confirmation messages sent back to the home (e.g., "ownership change" in DASH [26]), and also eliminates the associated protocol engine occupancy. Therefore, our protocol handles 3-hop write transactions involving a remote owner more efficiently. Second, we inherently eliminate livelock and starvation problems that arise due to the presence of NAKs. In contrast, the SGI Origin [25] uses a number of complicated mechanisms such as keeping retry counts and reverting to a strict request-reply protocol, while most other protocols with NAKs ignore this important problem (e.g, DASH [26], FLASH [24]).

We use a number of unique techniques to limit the amount of buffering needed in the network for avoiding deadlocks. First, the network uses ''hot potato'' routing with increasing age and priority when a message is non-optimally routed. This enables a message to theoretically reach an empty buffer anywhere in the network, making our buffering requirements grow linearly as opposed to quadratically with additional nodes. Second, the buffer space is shared among all lanes, so we do not need separate buffer space per lane. Third, we bound the number of messages injected in the network as a result of a single request. The key place where this is necessary is for invalidation messages. We have developed a new technique, called *cruise-missile-invalidates (CMI)*, that allows us to invalidate a large number of nodes by injecting only a handful of invalidation messages into the network. Each invalidation message visits a predetermined set of nodes, and eventually generates a single acknowledgment message when it reaches the final node in that set. Our studies show that CMI can also lead to superior invalidation latencies by avoiding serializations that arise from injecting many invalidation messages from the home node and gathering the corresponding acknowledgments at the requesting node. The above properties allow us to provide a limited amount of buffering per node that does not need to grow as we add more nodes.[4]

2.6 System Interconnect

The Piranha system interconnect consists of three distinct components: the *output queue (OQ)*, the *router (RT)* and the *input queue (IQ)*. The OQ accepts packets via the packet switch from the protocol engines or from the system controller. The RT transmits and receives packets to and from other nodes, and also deals with transit traffic that passes through the RT without impacting other modules. The IQ receives packets that are addressed to the local node and forwards them to the target module via the packet switch.

The interconnect system can also be used to initialize Piranha chips. This method relies on the RT to initialize channels automatically. By default (after reset), the RT forwards all initialization packets to the system controller (SC), which interprets control packets and can access all control registers on a Piranha node. Other SC capabilities related to initialization include accessing the on-chip memories, updating the routing table, starting/stopping individual Alpha cores, and testing the off-chip memory. Piranha can also be initialized using the traditional Alpha boot process, where the primary caches are loaded from a small external EPROM over a bit-serial connection.

2.6.1 The Router (RT)

The RT is similar to the S-Connect design developed for the S3.mp project [30]. Like the S-Connect, the RT uses a topology-independent, adaptive, virtual cut-through router core based on a common buffer pool that is shared across multiple priorities and virtual channels. Since Piranha nodes are not separated by long distances, we do not use in-band clock distribution and synchronization mechanisms as in the S-Connect. Furthermore, Piranha links are nearly 50 times faster than S-Connect links, hence the internal structure of our router is more advanced.

Each Piranha processing node has four channels that are used to connect it to other nodes in a point-to-point fashion. Each I/O node has two channels, allowing it to be connected to two other nodes for redundancy. The system interconnect supports two distinct packet types. The *Short* packet format is 128 bits long and is used for all data-less transactions. The *Long* packet has the same 128-bit header format along with a 64 byte (512 bit) data section. Packets are transferred in either 2 or 10 interconnect clock cycles.

Each interconnect channel consists of two sets of 22 wires, one set for each direction. These wires are high-quality transmission lines that are driven by special low-voltage swing CMOS drivers and are terminated on-chip at the remote end by matching receivers. The signaling rate is four times the system clock frequency, or 2 Gbits/sec per wire. With four channels, each Piranha processing node has a total interconnect bandwidth of 32GB/sec. Channels use a piggyback handshake mechanism that deals with flow-control and transmission error recovery. Piranha uses a DC-balanced encoding scheme to minimize electrical problems related to high-speed data transmission. By guaranteeing that 11 of the 22 wires will always be in the '1' state while the others are in the '0' state, the net current flow along a channel is zero. This also allows a reference voltage for differential receivers to be generated at the termination without doubling the number of signal wires. The signaling scheme encodes 19 bits into a 22-bit DC-balanced word. Piranha sends 16 data bits along with 2 extra bits that are used for CRC, flow control and error recovery. By design, the set of codes used to represent 18 bits has no two elements that are complementary. This allows the 19^{th} bit, which is generated randomly, to be encoded by inverting all 22 bits. The resulting code is inversion insensitive and it DC-balances the links statistically in the time-domain along each wire. Therefore Piranha can use fiber-optic ribbons to interconnect nodes, as well as transformer coupling to minimize EMI problems for cables connecting two Piranha boxes.

2.6.2 The Input (IQ) and Output (OQ) Queues

The OQ provides a modest amount of buffering through a set of FIFOs that de-couple the operation of the router from the local node. The fall-through path is optimized, with a single cycle delay when the router is ready for new traffic. However, as the interconnect load increases, the router gives priority to transit traffic, and accepts new packets only when it has free buffer space and no incoming packets. This policy results in better overall performance. The OQ also supports 4 priority levels and ensures that lower priority packets cannot block higher priority traffic. This property is maintained throughout the system interconnect.

The IQ receives packets from the RT and forwards them to their target modules via the packet switch. It is important to quickly remove terminal packets from the RT because the high

3. Our protocol needs to support only a single forwarded request per request that is outstanding from the owner node. Therefore, we can use the TSRF entry allocated for the outstanding request to save information about the delayed forwarded request.
4. For example, with 16 TSRF entries per protocol engine and the use of CMI to limit invalidation messages to a total of 4, buffering for a total of 128 message headers (2 protocol engines * 16 TSRFs * 4 invalidations) is needed at each node with only 32 of them requiring space for data. Note that this buffer size is not a function of the number of nodes in the system!

speed operation makes buffering in the RT expensive. For this reason, the IQ has more buffer space than the OQ. Like the OQ, the IQ supports four priority levels. To improve overall system performance, the IQ allows low priority traffic to bypass high priority traffic if the latter is blocked and the former can proceed to its destination.

The IQ is more complex than the OQ because it must interpret packets to determine their destination module. This process is controlled by a disposition vector that is indexed by the packet type field (4 bits encode 16 major packet types). During normal operation, most packets are directed at the protocol engines while some packets (e.g., interrupts) are delivered to the system controller.

2.7 Reliability Features

Piranha supports a number of elementary *Reliability, Availability, and Serviceability (RAS)* features such as redundancy on all memory components, CRC protection on most datapaths, redundant datapaths, protocol error recovery[5], error logging, hot-swappable links, and in-band system reconfiguration support. Furthermore, Piranha attempts to provide a platform for investigating advanced RAS features for future large-scale servers. Although developing complete solutions for RAS in large-scale systems is beyond the scope of the project, our design provides hardware hooks to enable future research in this area. These RAS features can be implemented by changing the semantics of memory accesses through the flexibility available in the programmable protocol engines.

Examples of RAS features of interest are persistent memory regions, memory mirroring, and dual-redundant execution. Persistent memory regions can survive power failures, system crashes or other transient errors, and can greatly accelerate database applications that currently rely on committing state to disk or NVDRAM at transaction boundaries. Beyond adding a battery to the main memory banks and designing the memory controller so that it can power cycle safely, persistent memory requires mechanisms to force volatile (cached) state to safe memory, as well as mechanisms to control access to persistent regions. This can be implemented by making the protocol engines intervene in accesses to persistent areas and perform capability checks or persistent memory barriers. Similarly, Piranha's protocol engines can be programmed to intervene on memory accesses to provide automatic data mirroring, or to perform checks on the results of dual-redundant computation.

3 Evaluation Methodology

This section describes the workloads, simulation platform, and various architectures that are used in this study.

3.1 Workloads

Our OLTP workload is modeled after the TPC-B benchmark [43]. This benchmark models a banking database system that keeps track of customers' account balances, as well as balances per branch and teller. Each transaction updates a randomly chosen account balance, which includes updating the balance of the branch the customer belongs to and the teller from which the transaction is submitted. It also adds an entry to the history table, which keeps a record of all submitted transactions. Our DSS workload is modeled after Query 6 of the TPC-D benchmark [44]. The TPC-D benchmark represents the activities of a business that sells a large number of products on a worldwide scale. It consists of several inter-related tables that keep information such as parts and customer orders. Query 6 scans the largest table in the database to assess the increase in revenue that would have resulted if some discounts were eliminated. The behavior of this query is representative of other TPC-D queries [4], though some queries exhibit less parallelism.

We use the Oracle 7.3.2 commercial database management system as our database engine. In addition to the server processes that execute the actual database transactions, Oracle spawns a few daemon processes that perform a variety of duties in the execution of the database engine. Two of these daemons, the database writer and the log writer, participate directly in the execution of transactions. The database writer daemon periodically flushes modified database blocks that are cached in memory out to disk. The log writer daemon is responsible for writing transaction logs to disk before it allows a server to commit a transaction.

Our OLTP and DSS workloads are set up and scaled in a similar way to a previous study that validated such scaling [4]. We use a TPC-B database with 40 branches with a shared-memory segment (SGA) size of approximately 600MB (the size of the metadata area is about 80MB). Our runs consist of 500 transactions after a warm-up period. We use Oracle in a dedicated mode for this workload, whereby each client process has a dedicated server process for serving its transactions. To hide I/O latencies, including the latency of log writes, OLTP runs are usually configured with multiple server processes per processor. We use 8 processes per processor in this study. For DSS, we use Oracle with the Parallel Query Optimization option, which allows the database engine to decompose the query into multiple sub-tasks and assign each one to an Oracle server process. The DSS experiments use an in-memory 500MB database, and the queries are parallelized to generate four server processes per processor.

3.2 Simulation Environment

For our simulations, we use the SimOS-Alpha environment (the Alpha port of SimOS [37]), which was used in a previous study of commercial applications and has been validated against Alpha multiprocessor hardware [4]. SimOS-Alpha is a full system simulation environment that simulates the hardware components of Alpha-based multiprocessors (processors, MMU, caches, disks, console) in enough detail to run Alpha system software. Specifically, SimOS-Alpha models the micro-architecture of an Alpha processor [10] and runs essentially unmodified versions of Tru64 Unix 4.0 and PALcode.

The ability to simulate both user and system code under SimOS-Alpha is essential given the rich level of system interactions exhibited by commercial workloads. For example, for the OLTP runs in this study, the kernel component is approximately 25% of the total execution time (user and kernel). In addition, setting up the workload under SimOS-Alpha is particularly simple since it uses the same disk partitions, databases, application binaries, and scripts that are used on our hardware platforms to tune the workload.

SimOS-Alpha supports multiple levels of simulation detail, enabling the user to choose the most appropriate trade-off between simulation detail and slowdown. The fastest simulator uses an on-the-fly binary translation technique similar to Embra [48] to position the workload into a steady state. For the medium-speed (in simulation time) processor module, SimOS-Alpha models a single-issue pipelined processor. Finally, the slowest-speed processor module models a multiple-issue out-of-order processor. We use the medium-speed in-order model for evaluating the Piranha processor cores and the slow-speed out-of-order model to evaluate aggressive next-generation processors.

5. The TSRF associated with each protocol transaction maintains its state and keeps track of expected replies. The protocol engines can monitor for failures via mechanisms such as time-outs and error messages. When necessary, such state can be encapsulated in a control message and directed to recovery or diagnostic software.

Parameter	Piranha (P8)	Next-Generation Microprocessor (OOO)	Full-Custom Piranha (P8F)
Processor Speed	500 MHz	1 GHz	1.25 GHz
Type	in-order	out-of-order	in-order
Issue Width	1	4	1
Instruction Window Size	-	64	-
Cache Line Size	64 bytes	64 bytes	64 bytes
L1 Cache Size	64 KB	64 KB	64 KB
L1 Cache Associativity	2-way	2-way	2-way
L2 Cache Size	1 MB	1.5 MB	1.5 MB
L2 Cache Associativity	8-way	6-way	6-way
L2 Hit / L2 Fwd Latency	16 ns / 24 ns	12 ns / NA	12 ns / 16 ns
Local Memory Latency	80 ns	80 ns	80 ns
Remote Memory Latency	120 ns	120 ns	120 ns
Remote Dirty Latency	180 ns	180 ns	180 ns

Table 1. Parameters for different processor designs.

3.3 Simulated Architectures

Table 1 presents the processor and memory system parameters for the different processor configurations we study. For our next-generation microprocessor, we model a very aggressive design similar to Alpha 21364 which integrates a 1GHz out-of-order core, two levels of caches, memory controller, coherence hardware, and network router all on a single die (with a comparable area to Piranha's processing chip). The use of an ASIC process limits the frequency of the processor cores in Piranha to 500 MHz. In addition, the use of the lower density ASIC SRAM cells, along with the integration of eight simple processor cores, limits the amount of second-level on-chip cache in Piranha. However, the lower target clock frequency in Piranha allows for a higher associativity cache. The full-custom Piranha parameters are used to illustrate the potential for the Piranha architecture if the design were to be done with a larger team and investment. Given the simple single-issue in-order pipeline, it is reasonable to assume that a full-custom approach can lead to a faster clock frequency than a 4-issue out-of-order design.

Table 1 also shows the memory latencies for different configurations. Due to the lack of inclusion in Piranha's L2 cache, there are two latency parameters corresponding to either the L2 servicing the request (L2 Hit) or the request being forwarded to be serviced by another on-chip L1 (L2 Fwd). As shown in Table 1, the Piranha prototype has a higher L2 hit latency than a full-custom processor due to the use of slower ASIC SRAM cells.

4 Performance Evaluation of Piranha

This section compares the performance of Piranha with an aggressive out-of-order processor (OOO in Table 1) in both single-chip and multi-chip configurations. In addition, we present results for a potential full-custom Piranha design (P8F in Table 1) that more fairly judges the merits of the architecture. We use the OLTP and DSS database workloads described in the previous section for this evaluation.

Figure 5 shows our results for single-chip configurations for both OLTP and DSS. We study four configurations: a hypothetical single-CPU Piranha chip (*P1*), a next-generation out-of-order processor (*OOO*), a hypothetical single-issue in-order processor otherwise identical to OOO (*INO*), and the actual eight-CPU Piranha chip (*P8*). The P1 and INO configurations are used to better isolate the various factors that contribute to the performance

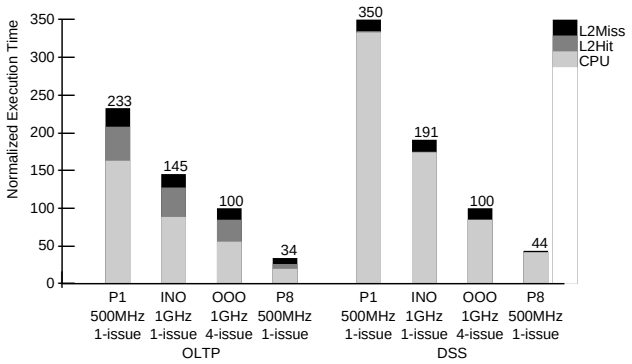
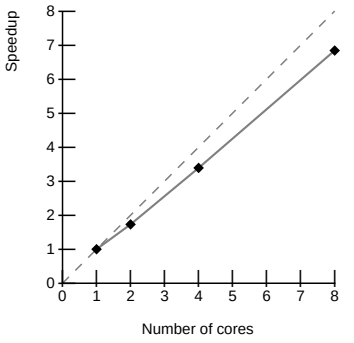


Figure 5. Estimated performance of a single-chip Piranha (8 CPUs/chip) versus a 1GHz out-of-order processor.

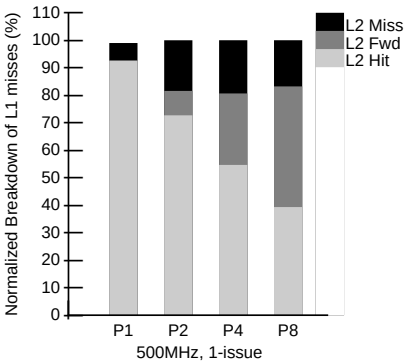
differences between OOO and P8. The figure shows execution time normalized to that of OOO. The execution time is divided into CPU busy time, L2 hit stall time, and L2 miss stall time. For the P8 configuration, the L2 hit stall time includes both L2 hits as well as forwarded L2 requests served by an L1 (see L2 Fwd latency in Table 1). Focusing on the OLTP results, we observe that OOO outperforms P1 (as expected) by about 2.3 times. The INO result shows that the faster frequency (1GHz vs. 500MHz) and lower L2 hit latency (12ns in INO/OOO vs. 16/24ns in P1/P8) alone account for an improvement of 1.6 times. The wider-issue and out-of-order features provide the remaining 1.45 times gain. However, once we integrate eight of the simple CPUs, the single-chip Piranha (P8) outperforms OOO by almost 3 times.

As shown in Figure 6(a), the reason for Piranha's exceptional performance on OLTP is that it achieves a speedup of nearly seven times with eight on-chip CPUs relative to a single CPU (P1). This speedup arises from the abundance of thread-level parallelism in OLTP, along with the extremely tight-coupling of the on-chip CPUs through the shared second-level cache (leading to small communication latencies), and the effectiveness of the on-chip caches in Piranha. The last effect is clearly observed in Figure 6(b) which shows the behavior of the L2 cache as more on-chip CPUs are added. This figure shows a breakdown of the total number of L1 misses that are served by the L2 (L2 Hit), forwarded to another on-chip L1 (L2 Fwd), or served by the memory (L2 Miss). Although the fraction of L2 hits drops from about 90% to under 40% when we go from 1 to 8 CPUs, the fraction of L2 misses that go to memory remains constant at under 20% past a single CPU. In fact, adding CPUs (and their corresponding L1s) in Piranha's non-inclusive cache hierarchy actually increases the amount of on-chip memory (P8 doubles the on-chip memory compared to P1), which partially offsets the effects of the increased pressure on the L2. The overall trend is that as the number of CPUs increases, more L2 misses are served by other L1s instead of going to memory. Even though "L2 Fwd" accesses are slower than L2 Hits (24ns vs. 16ns), they are still much faster than a memory access (80ns). Overall, Piranha's non-inclusion policy is effective in utilizing the total amount of on-chip cache memory (i.e., both L1 and L2) to contain the working set of a parallel application.

In addition to the above on-chip memory effects, the simultaneous execution of multiple threads enables Piranha to tolerate long latency misses by allowing threads in other CPUs to proceed independently. As a result, a Piranha chip can sustain a relatively high CPU utilization level despite having about 3x the number of L2 misses compared to OOO (from simulation data not shown here). On-chip and off-chip bandwidths are also not a problem even with eight CPUs because OLTP is primarily latency bound. Finally, OLTP workloads have been shown to exhibit constructive



(a)



(b)

Figure 6. Piranha's (a) speedup and (b) L1 miss breakdown for OLTP.

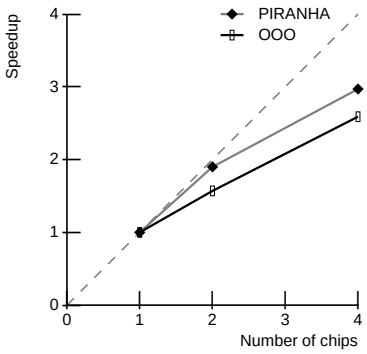


Figure 7. Speedup of OLTP in multi-chip systems with 500 MHz 4-CPU Piranha chips versus 1GHz out-of-order chips. (A single-chip 4-CPU Piranha is approximately 1.5x faster than the single-chip OOO).

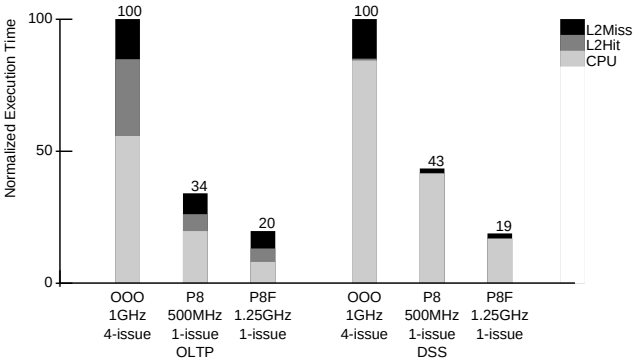


Figure 8. Performance potential of a full-custom Piranha chip for OLTP and DSS.

interference in the instruction and data streams [27], and this works to the benefit of Piranha.

Piranha's performance edge over OOO in transaction processing is robust to the specific workload used and to changes in design parameters. Using a workload modeled after the TPC-C benchmark [45], our results showed that P8 outperforms OOO by over a factor of 3 times. We also studied the sensitivity of Piranha's performance to more pessimistic design parameters: 400MHz CPUs with 32KB one-way L1s, and L2 latencies of 22ns (L2 Hit) and 32ns (L2 Fwd). Even though the execution time increases by 29% with these parameters, Piranha still holds a 2.25 times performance advantage over OOO on OLTP.

Referring back to Figure 5, we see that Piranha (P8) also outperforms OOO for DSS, although by a narrower margin than for OLTP (2.3 times). The main reason for the narrower margin comes from the workload's smaller memory stall component (under 5% of execution time) and better utilization of issue slots in a wide-issue out-of-order processor. DSS is composed of tight loops that exploit spatial locality in the data cache and have a smaller instruction footprint than OLTP. Since most of the execution time in DSS is spent in the CPU, OOO's faster clock speed alone nearly doubles its performance compared to P1 (P1 vs. INO), with almost another doubling due to wider-issue and out-of-order execution (INO vs. OOO). However, the smaller memory stall component of DSS also benefits Piranha, as it achieves near-linear speedup with 8 CPUs (P8) over a single CPU (P1).

One interesting alternative to consider for Piranha is to trade CPUs for a larger L2 cache. However, since the fraction of L2 miss stall time is relatively small (e.g., about 22% for P8 in Figure 5), the improvement in execution time from even an infinite L2 would also be modest. Moreover, since Piranha CPUs are small, relatively little SRAM can be added per CPU removed. As a result, such a trade-off does not seem advantageous for Piranha. There is however a relatively wide design space if one considers increasingly complex CPUs in a chip-multiprocessing system. A thorough analysis of this trade-off is beyond the scope of this paper.

In addition to the single-chip comparisons above, it is important to evaluate how a Piranha system performs in multi-chip (i.e., NUMA) configurations. Figure 7 shows the speedup trends for OLTP when going from a single chip to a four-chip system for both Piranha and OOO (DSS scalability, not shown, is near linear for both systems). In these experiments, the Piranha chip uses 4 CPUs per chip[6] (i.e., P4). The figure shows that the Piranha system scales better than OOO (3.0 vs. 2.6) for the range of system sizes studied. This is somewhat surprising, since operating system scalability limitations could adversely affect Piranha given its higher total count of 16 (albeit slower) CPUs versus 4 for OOO. However, we observe that the effectiveness of on-chip communication in Piranha offsets the OS overheads normally associated with larger CPU counts. In general we expect Piranha system scalability to be on par with that of OOO systems.

So far we have considered the performance of Piranha under the constraints of the ASIC design methodology being used to implement the prototype. To fairly judge the potential of the Piranha approach, we also evaluate the performance of a full-

6. The current version of the operating system that we use in our simulation environment limits us to 16 CPUs. Therefore, to study multi-chip scaling, we consider Piranha chips with four on-chip CPUs.

custom implementation (see Table 1 for P8F parameters). Figure 8 compares the performance of a full-custom Piranha with that of OOO, both in single-chip configurations. The figure shows the faster full-custom implementation can further boost Piranha's performance to 5.0 times over OOO in OLTP and 5.3 times in DSS. DSS sees particularly substantial gains since its performance is dominated by CPU busy time, and therefore it benefits more from the 150% boost in clock speed (P8 vs. P8F). The gains in OLTP are also mostly from the faster clock cycle, since the relative improvement in memory latencies is smaller with respect to the original P8 parameters.

Overall, the Piranha architecture seems to be a better match for the underlying thread-level parallelism available in database workloads than a typical next generation out-of-order superscalar processor design which relies on its ability to extract instruction-level parallelism.

5 Design Methodology and Implementation Status

Our design methodology starts with architectural specification in the form of C++ based models for each of the major Piranha modules (e.g., L2 cache, protocol engine). The C++ models implement behavior in a cycle-accurate fashion and use the same boundary signals as in the actual implementation. These models form the starting point for Verilog coding. We have currently completed a first pass of the Verilog for the processor core and are doing initial synthesis for timing. The remaining modules are at different phases of C++ or Verilog development. The C++ models execute much faster than their Verilog counterparts, allowing for more efficient functional and architectural verification. Our environment also allows C++ and Verilog models to be interchanged or mixed for development and verification purposes. Finally, the coherence protocols will also be verified using formal methods.

Piranha is being implemented in a semi-custom 0.18 micron ASIC design flow [19]. This design flow uses industry standard hardware description languages and synthesis tools. Hence, it has the advantage of improved portability to evolving ASIC process technologies and shorter time-to-market when compared to full-custom design methodologies. To achieve our target 500 MHz frequency, we depend on a small number of custom circuit blocks for some of our time-critical SRAM cache memory, and use a few specialized synthesis and layout tools that specifically target datapaths and arithmetic units. The ASIC process technology includes high density SRAM with cell sizes on the order of 4.2 μm^2 [6] and gate delays of 81ps (worst case) for an unloaded 2-input NAND.

Although the entire Piranha implementation is not complete, we can infer the clock frequency from preliminary logic synthesis of the processor core and critical path estimates for the various modules. We have also calculated the area for each of the major modules using estimates from compilable memory arrays, logic synthesis, and simple gate counts. From these area estimates, we have developed a general floor-plan of the Piranha processing node illustrated in Figure 9. Roughly 75% of the Piranha processing node area is dedicated to the Alpha cores and L1/L2 caches, with the remaining area allocated to the memory controllers, intra-chip interconnect, router, and protocol engines.

6 Discussion and Related Work

In addition to chip multiprocessing (CMP), the Piranha project incorporates other interesting ideas in the general area of scalable shared-memory designs. Much of the related work has already been referenced in earlier sections. We further discuss some of the previous work pertinent to database workloads and CMP in this section.

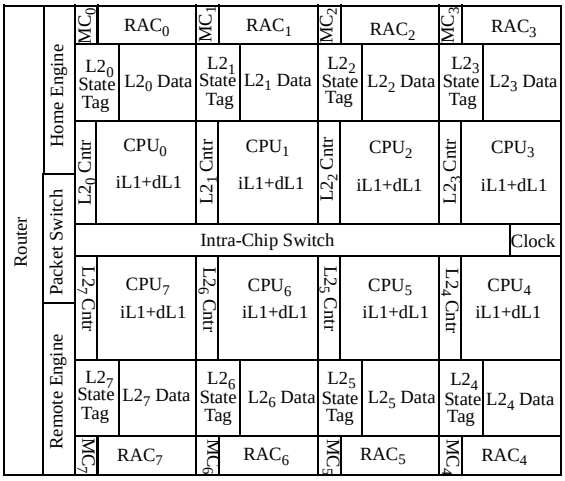


Figure 9. Floor-plan of the Piranha processing node with eight CPU cores.

There have been a large number of recent studies of database applications (both OLTP and DSS) due to the increasing importance of these workloads [4,7,8,12,21,27,28,34,35,36,42,46]. To the best of our knowledge, this is the first paper that provides a detailed evaluation of database workloads in the context of chip multiprocessing. Ranganathan et al. [35] study user-level traces of database workloads in the context of wide-issue out-of-order processors, and show that the gains for DSS are substantial while the gains for OLTP are more limited (consistent with our results in Section 4). A number of studies address issues related to the effectiveness of different memory system architectures for OLTP workloads. Barroso et al. [4] show the need for large direct-mapped off-chip caches (8 MB). Lo et al. [27] show that a large off-chip cache (16 MB) is not adversely affected by cache interference caused by fine-grain multithreading. A more recent study shows that smaller, more associative caches (e.g., 2MB 4-way) that can be integrated on-chip can actually outperform larger direct-mapped off-chip caches [3]. Our results here show that small associative second-level on-chip caches (1MB 8-way in our case) are still effective when shared among multiple processors or threads. Finally, Barroso et al. [3] show that aggressive chip-level integration of the memory system, coherence, and network modules on a single chip (as in Alpha 21364) can provide large gains for OLTP workloads.

Piranha advocates a focused design that targets commercial applications (which currently constitute the largest segment for high-performance servers) at the possible expense of other types of workloads. There are several other contemporary processor designs that are specifically focused on commercial markets [5,23].[7]

Several papers from Stanford have advocated and evaluated the use of chip multiprocessing (CMP) in the context of workloads such as SPEC [15,29,33], and the Hydra project is exploring CMP with a focus on thread-level speculation [16,17]. The current implementation integrates four 250MHz processors each with 8KB instruction and data caches and a shared 128KB second-level cache onto a small chip. There are a number of differences between Hydra and Piranha. For example, Piranha has eight cores, a second-level cache which does not maintain inclusion, a high-speed switch instead of a bus to connect the on-chip cores, and provides scalability past a single chip by integrating the required on-chip function-

7. The latest IBM RS-III 450MHz processor currently holds the TPC-C benchmark record (models OLTP) with 24 processors, outperforming some systems with over 64 processors (see http://www.tpc.org).

ality to support glueless multiprocessing. Furthermore, Piranha focuses on commercial workloads, which have an abundance of explicit thread-level parallelism. Therefore, support for thread-level speculation as proposed by Hydra and others [22,41] is not necessary for achieving high performance on such workloads.

Another CMP design in progress is the IBM Power4 [9]. Each Power4 chip has two 1-GHz, five-issue, out-of-order superscalar processor cores, along with an on-chip shared L2 cache. Four such chips can be connected on a multi-chip module to form an eight processor system with a logically shared L2 cache. The information from IBM does not elaborate on the expansion capabilities past four chips. Piranha takes a more extreme approach by incorporating eight much simpler processor cores on a single chip, and provides on-chip functionality for a scalable design. Finally, Sun Microsystems has also announced a new CMP design called the MAJC 5200 [47], which is the first implementation of the MAJC architecture targeted at multimedia and Java applications. The 5200 contains two 500MHz VLIW processors, each capable of issuing four instructions per cycle. The cores each have their own 16KB instruction cache, but share a 16KB, 4-way L1 data cache. The choice of sharing the L1 cache clearly does not scale well to more cores. Furthermore, the small size of the L1 along with the lack of an on-chip L2 cache makes this design non-optimal for commercial workloads such as OLTP.

Simultaneous multithreading (SMT) [11] (and other forms of multithreading) is an alternative to CMP for exploiting the thread-level parallelism in commercial workloads. In fact, Lo et al. [27] have shown that SMT can provide a substantial gain for OLTP workloads and a reasonably large gain for DSS workloads when it is coupled with very wide-issue out-of-order processors. An SMT processor adds extra functionality and resources (e.g., larger register file) to an out-of-order core to support multiple simultaneous threads. As such, SMT increases the implementation and verification complexity that comes with such designs. Furthermore, intelligent software resource management is sometimes necessary in SMT to avoid negative performance effects due to the simultaneous sharing of critical resources such as the physical register file, L1 caches, and TLBs [27]. The advantage of SMT over CMP is that it provides superior performance on workloads that do not exhibit thread-level parallelism. Because the Piranha design targets workloads with an abundance of parallelism, we have opted to forgo single-thread performance in favor of design simplicity.

Our evaluation of Piranha has primarily focused on commercial database workloads. We expect Piranha to also be well suited for a large class of web server applications that have explicit thread-level parallelism. Previous studies have shown that some web server applications, such as the AltaVista search engine, exhibit behavior similar to decision support (DSS) workloads [4].

7 Concluding Remarks

The use of chip multiprocessing is inevitable in future microprocessor designs. Advances in semiconductor technology are enabling designs with several hundred million transistors in the near future. Next-generation processors such as the Alpha 21364 are appropriately exploiting this trend by integrating the complete cache hierarchy, memory controllers, coherence hardware, and network routers all onto a single chip. As more transistors become available, further increasing on-chip cache sizes or building more complex cores will only lead to diminishing performance gains and possibly longer design cycles in the case of the latter option. While techniques such as simultaneous multithreading can remedy the diminishing gains, they do not address the increasing design complexity. At the same time, using the extra transistors to integrate multiple processors onto the same chip is quite promising, especially given the abundance of explicit thread-level parallelism in important commercial workloads. At least a couple of next-generation processor designs subscribe to this philosophy by integrating two superscalar cores on a single die. The key questions for designers of future processors will not be whether to use chip multiprocessing, but the appropriate trade-off between the number of cores and the power of each core, and how to best partition the memory hierarchy among the multiple cores.

This paper described the Piranha architecture which takes an extreme position on chip multiprocessing (CMP) by integrating eight simple processor cores along with a complete cache hierarchy, memory controllers, coherence hardware, and network router all onto a single chip to be built with the next-generation 0.18um CMOS process. Due to our small design team and the modest investment in this research prototype, we opted for an ASIC design with simple single-issue in-order processor cores. Even with this handicap, our results show that Piranha can outperform aggressive next-generation processors by a factor of 2.9 times (on a per chip basis) on important commercial workloads such as OLTP. A full-custom design, which would require a larger design team, has the potential to extend this performance advantage to almost five times. Our results clearly indicate that focused designs such as Piranha that directly target commercial server applications can substantially outperform general-purpose microprocessor designs with much higher complexity. On the other hand, Piranha is the wrong design choice if the goal is to achieve the best SPECint or SPECfp numbers because of the lack of sufficient thread-level parallelism in such workloads.

We hope that our experience in building the Piranha prototype provides a proof point for CMP designs based on simple processor cores. We also hope that some of the design options we are exploring, such as the lack of inclusion in the shared second-level cache, the interaction between the intra-node and inter-node coherence protocols, the efficient inter-node protocol, and the unique I/O architecture, provide further insight for future CMP processors and scalable designs in general.

Acknowledgments

Several people have contributed to the Piranha effort and to the preparation of this manuscript. We would like to thank Gary Campbell for his sponsorship of this project, Alan Eustace and Bob Iannucci for their continuing support, and Marco Annaratone and Bob Supnik for their early support of the idea. Bill Bruckert, Harold Miller, and Jim Whatley have been key in steering Piranha toward a real design effort. The following people have also made significant technical contributions to Piranha: Joan Pendleton wrote the initial Verilog for the Alpha core, Dan Scales helped with the inter-chip coherence protocol, Basem Nayfeh was an early member of the architecture team, Robert Bosch developed the SimOS-Alpha out-of-order processor model, and Jeff Sprouse has helped with the Verilog development. Technical discussions with Pete Bannon, John Edmonson, Joel Emer, and Rick Kessler were important in focusing our effort and refining our strategy. We are grateful to Keith Farkas, Jeff Mogul, Dan Scales, and Deborah Wallach for their careful review of the manuscript. Finally, we would like to thank the anonymous reviewers for their comments.

References

[1] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *15th Annual International Symposium on Computer Architecture*, pages 280-289, May 1988.

[2] P. Bannon. Alpha 21364: A Scalable Single-chip SMP. Presented at the *Microprocessor Forum '98 (http://www.digital.com/alpha-oem/microprocessorforum.htm)*, October 1998.

[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 Ar-*

chitecture, 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] J. Borkenhagen and S. Storino. 5th Generation 64-bit PowerPC-Compatible Commercial Processor Design. *http://www.rs6000.ibm.com /resource/technology/pulsar.pdf*. September 1999.

[6] S. Crowder et al. IEDM Technical Digest, page 1017, 1998.

[7] Z. Cvetanovic and D. Bhandarkar. Characterization of Alpha AXP Performance using TP and SPEC Workloads. In *21st Annual International Symposium on Computer Architecture*, pages 60–70, April 1994.

[8] Z. Cvetanovic and D. Donaldson. AlphaServer 4100 Performance Characterization. In *Digital Technical Journal*, 8(4), pages 3-20, 1996.

[9] K. Diefendorff. Power4 Focuses on Memory Bandwidth: IBM Confronts IA-64, Says ISA Not Important. In *Microprocessor Report*, Vol. 13, No. 13, October 1999.

[10] Digital Equipment Corporation. Digital Semiconductor 21164 Alpha Microprocessor Hardware Reference Manual. March 1996.

[11] 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.

[12] R. J. Eickemeyer, R. E. Johnson, S. R. Kunkel, M. S. Squillante, and S. Liu. Evaluation of Multithreaded Uniprocessors for Commercial Application Environments. In *23rd Annual International Symposium on Computer Architecture*, pages 203–212, May 1996.

[13] J. S. Emer. Simultaneous Multithreading: Multiplying Alpha's Performance. Presentation at the *Microprocessor Forum '99*, October 1999.

[14] A. Gupta, W.-D. Weber, and T. Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. In *International Conference on Parallel Processing*, July 1990.

[15] L. Hammond, B. Nayfeh, and K. Olukotun. A Single-Chip Multiprocessor. In *IEEE Computer* 30(9), pages 79-85, September 1997.

[16] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *8th ACM International Symposium on Architectural Support for Programming L anguages and O perating Systems*, San Jose, California, October 1998.

[17] L. Hammond, B. Hubbert, M. Siu, M. Prabhu, M. Willey, M. Chen, M. Kozyrczak, and K. Olukotun. The Stanford Hydra CMP. Presented at *Hot Chips 11*, August 1999.

[18] J. Hennessy. The Future of Systems Research. In *IEEE Computer*, Vol. 32, No. 8, pages 27-33, August 1999.

[19] IBM Microelectronics. ASIC SA27E Databook. International Business Machines, 1999.

[20] N. P. Jouppi and S. Wilton. Tradeoffs in Two-Level On-Chip Caching. In *21st Annual International Symposium on Computer Architecture*, pages 34-45, April 1994.

[21] 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.

[22] V. Krishnan and J. Torrellas. Hardware and Software Support for Speculative Execution of Sequential Binaries on Chip-Multiprocessor. In *ACM International Conference on Supercomputing (ICS'98)*, pages 85-92, June 1998.

[23] S. Kunkel, B. Armstrong, and P. Vitale. System Optimization for OLTP Workloads. *IEEE Micro*, Vol. 19, No. 3, May/June 1999.

[24] J. Kuskin et al. The Stanford FLASH Multiprocessor. In *21st Annual International Symposium on Computer Architecture*, April 1994.

[25] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *24 th Annual International Symposium on Computer Architecture*, pages 241-251, June 1997.

[26] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. L. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *17 th Annual International Symposium on Computer Architecture*, pages 94-105, May 1990.

[27] 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.

[28] A. M. G. Maynard, C. M. Donnelly, and B. R. Olszewski. Contrasting Characteristics and Cache Performance of Technical and Multi-User Commercial Workloads. In *6th International Conference on Architectural Support for Programming L anguages and O perating Systems* pages 145–156, October 1994.

[29] B. Nayfeh, L. Hammond, and K. Olukotun. Evaluation of Design Alternatives for a Multiprocessor Microprocessor. In *23rd Annual International Symposium on Computer Architecture*, May 1996.

[30] A. Nowatzyk, G. Aybay, M. Browne, W. Radke, and S. Vishin. S-Connect: from Networks of Workstations to Supercomputing Performance. In *22nd Annual International Symposium on Computer Architecture*, pages 71-82, May 1995.

[31] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, W. Radke, and S. Vishin. The S3.mp Scalable Shared Memory Multiprocessor. In *International Conference on Parallel Processing (ICPP'95)*, pages I.1 - I.10, July 1995.

[32] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, W. Radke, and S. Vishin. Exploiting Parallelism in Cache Coherency Protocol Engines. In *EuroPar'95 International Conference on Parallel Processing*, August 1995.

[33] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K.-Y. Chang. The Case for a Single-Chip Multiprocessor. In *7 th International Symposium on Architectural Support for Programming L anguages and O perating Systems*, October 1996.

[34] S. E. Perl and R. L. Sites. Studies of Windows NT Performance Using Dynamic Execution Traces. In *2nd Symposium on O perating System Design and Implementation*, pages 169–184, October 1996.

[35] 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 L anguages and O perating Systems*, pages 307-318, October 1998.

[36] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. The Impact of Architectural Trends on Operating System Performance. In *15th Symposium on O perating System Principles*, December 1995.

[37] 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.

[38] A. Saulsbury, F. Pong, and A. Nowatzyk. Missing the Memory Wall: The Case for Processor/Memory Integration. In *23rd Annual International Symposium on Computer Architecture*. May 1996.

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

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

[41] J. Steffan and T. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *4 th International Symposium on High-Performance Computer Architecture*, pages 2-13, February 1998.

[42] S. S. Thakkar and M. Sweiger. Performance of an OLTP Application on Symmetry Multiprocessor System. In *17 th Annual International Symposium on Computer Architecture*, pages 228–238, May 1990.

[43] Transaction Processing Performance Council. TPC Benchmark B Standard Specification Revision 2.0. June 1994.

[44] Transaction Processing Performance Council. TPC Benchmark D (Decision Support) Standard Specification Revision 1.2. November 1996.

[45] Transaction Processing Performance Council. TPC Benchmark C, Standard Specification Revision 3.6, October 1999.

[46] P. Trancoso, J.-L. Larriba-Pey, Z. Zhang, and J. Torrellas. The Memory Performance of DSS Commercial Workloads in Shared-Memory Multiprocessors. In *3rd Annual International Symposium on High-Performance Computer Architecture*, pages 250–260, February 1997.

[47] M. Tremblay. MAJC-5200: A VLIW Convergent MPSOC. In *Microprocessor Forum*, October 1999.

[48] E. Witchel and M. Rosenblum. Embra: Fast and Flexible Machine Simulation. In *1996 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 68-79, May 1996.