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

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 (a joint effort by Corporate Research and Tandem Division ASIC Design Center). Piranha 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 Alpha processor cores in Piranha are based on a simple single-issue in-order pipelined design. Furthermore, we use an industry-standard ASIC approach for implementing our design. These design choices 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 2.8 times (on a per chip basis) on important workloads such as OLTP. The true potential of the Piranha architecture is more fairly judged if we consider a full-custom design which would require a larger team and investment but still maintains low design time and complexity. Our results show that such a full-custom design can extend Piranha’s performance advantage to up to five times over conventional designs. 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 spectacular performance gains on target applications such as the SPEC benchmark [41], continuing along this path is becoming less viable beyond the next generation of microprocessors. The microprocessor industry is currently plagued with higher development costs and longer design times. Available data from one commercial microprocessor company shows that within two processor generations, the design team grew by over five times, the verification effort was over two times larger, and the design took almost two and a half times longer [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 quickly 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 [4,7,8,22,29,35,37]. First, commercial workloads often lead to inefficient executions dominated by a large memory stall component. This behavior arises from large
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 [36]. 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. The next-generation processor technology will support well over a hundred million transistors, allowing designers to integrate functional modules that traditionally appear at the system-level onto the same die as the processor. For example, the next-generation Alpha 21364 plans to aggressively exploit such integration 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 a sufficiently aggressive existing 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 (given the diminishing returns from wider issue and more speculative execution).

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 users or clients, and has traditionally been used to hide the long latency of I/O operations in such workloads. Previous studies have shown that techniques such as simultaneous multithreading (SMT) can provide a substantial performance boost for database workloads [28]. In fact, the Alpha 21464 (successor to Alpha 21364) is planning to combine aggressive chip-level integration (as described in the 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 multiprocessoring (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 hierarchical 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 (a joint effort by Corporate Research and Tandem Division ASIC Design Center) to explore chip multiprocessoring architectures targeted at parallel commercial workloads. 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 the Piranha 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. These design choices allow us to build a complete system in two years with a team size and investment that are an order of magnitude smaller than that of a typical commodity processor. However, as a result of using an ASIC design methodology, 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.

---

1 The project name is motivated by the analogy of seemingly small fish that in concert can vanquish a much larger creature.
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 (previously known as Digital Unix). As expected, our results show that each Piranha processor core is substantially slower than an aggressive next-generation processor due to our use of a simple pipelined design implemented in an ASIC process. Nevertheless, the integration of eight cores onto a single chip allows Piranha to outperform next-generation processors by about 2.8 times on a per chip basis on important workloads, such as OLTP which have a large memory stall component, while providing less of an advantage on commercial workloads (such as decision support) which have a much smaller stall component.

The true potential of the Piranha architecture is more fairly judged by considering a full-custom design with more aggressive processor cores. 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 4.8 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.

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.

Piranha also has a unique I/O architecture. Aside from processing nodes, the Piranha design also includes an I/O node that integrates I/O device interfaces onto a single chip with a single processor core, a single memory controller, coherence protocol engines, and a simplified network router. Except for the I/O device interfaces, the remaining modules are identical in design to those used in the processing node. The I/O node is a full-fledged member of the interconnect and takes part in the global shared-memory coherence protocol. Furthermore, the ratio of processing nodes to I/O nodes is completely flexible and can be adjusted according to the needs of a particular workload. Finally, the presence of a processor core on the I/O chip provides several benefits; e.g., 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.

The rest of paper is structured as follows. Sections 2 and 3 present an overview of the Piranha architecture and our software strategy. We next describe our experimental methodology, including a brief description of the workloads and other architectures that we study. Section 5 presents detailed performance results for the Piranha prototype and considers more custom implementations that better illustrate the full potential of our approach. The design methodology and implementation status of the project are described in Section 6. 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, exception handling, performance monitoring, interrupt distribution). It should be noted that the various modules communicate exclusively through the connections shown in Figure 1, which also represent the actual signal connections. This modular approach leads to a strict hierarchical decomposition of the Piranha chip which allows 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. 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 the PCI/X module with the rest of the system. The dL1 module also provides the PCI/X with address translation facilities and provisions to access I/O space registers and to generate interrupts. 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 any mix of processing and I/O nodes. 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 system. In part, this decision is based on the observation that available inter-chip bandwidth is best invested in one general interconnect system that forms a global resource which can be dynamically utilized for both memory and I/O traffic. This approach also allows flexibility in the ratio of processing to I/O nodes.

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

### 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 [40]. It consists of a 500MHz pipelined datapath and contains hardware support for floating-point operations. The pipeline has been partitioned into 8-stages: instruction fetch, register-read, ALU 1 through 5, and write-back. The reason for the 5-stage ALU is to support pipelined floating-point and multiply instructions. Nevertheless, most instructions effectively execute in a single cycle. The processor core includes a number of 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 handling of synonyms is described in Section 2.3). The L1 cache modules include tag compare logic, instruction and data TLB’s (256 entries, 4-way associative), a store buffer (data cache only), and ICS interface logic. 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 just a central crossbar switch that interconnects most of the modules on a Piranha chip. However, given the fairly large number of ports to the crossbar switch, there are some challenging implementation problems, such as arbitration, flow control, and layout, that need be addressed. As mentioned before, the ICS is also the primary means of 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 a uni-directional, push-only interface. If module A needs to send a transaction to module B, it asserts a transaction request. Provided that module B has declared itself ready, the ICS will inform A that the transaction has been accepted through a grant signal and in the same cycle starts copying data from A to B. Therefore, data always flows from a requestor to the destination. As an example, a read request must be broken into two transactions: an address transaction that is sent to the target, and a separate data transaction that is initiated by the target when it has the data ready. Once the ICS commences a data transfer, it provides no further flow control. This implies that the source and destination of a transaction must be able to transfer data at a rate of one 64-bit word (plus an 8-bit error code) per cycle. Furthermore, once a transfer is commenced, the source and destination are ensured that no reordering will take place with respect to other transactions. These ordering properties of the ICS are exploited in supporting intra-chip coherence (described in Section 2.3). In order to maximize the available bandwidth, the ready signals to the ICS are actually pipelined so as to allow back-to-back transactions without dead cycles. In order to reduce latency, modules are allowed to issue the target destination of a future request ahead of the actual request. This hint is used by the ICS to pre-allocate data paths and to speculatively assert the requester’s grant signal.

The ICS is implemented by using a set of eight busses that run along the center of the Piranha chip. Each data path is 64+8 bits wide and can sustain one transfer per cycle. Given that the internal ICS capacity is 32 GB/sec or

---

2 This means that by the time a unit receives a single-cycle transaction it is too late to prevent a subsequent transaction to be sent. It must either provide enough buffering to absorb two consecutive transactions, de-assert its ready line speculatively (e.g., through toggling), or use higher-level protocol properties to assure that it will not receive back-to-back transactions.
about 3 times the available memory bandwidth, achieving an optimal schedule is not necessary for getting good performance. This provides us with some flexibility in our scheduling algorithms. Our current implementation statically partitions the resources into eight groups, leading to simpler arbitration logic.

Another important aspect of the ICS is that it supports multiple logical lanes that are used by the intra-chip cache coherence protocol to avoid protocol deadlocks. Instead of adding multiple datapaths for each module, multiple lanes are supported by simply giving each module multiple ready lines with distinct IDs. A requester can specify the appropriate lane for a transaction by using the corresponding ID for the destination. For example, each protocol engine has a low-priority ID used by new unsolicited requests and a high-priority ID used for sending solicited replies to the engine.

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. Each bank is 8-way set-associative with a 64B cache line size. A bank has its own control logic, private interface to a corresponding memory controller, and an ICS interface used for communication with all 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 the aggregate L1 capacity for our eight processor cores is 1MB, maintaining inclusion of the data in our 1MB L2 can potentially waste the full L2 capacity with duplicate data. Maintaining inclusion could also incur extraneous invalidations of L1 copies due to lack of sufficient associativity at the L2. Therefore, Piranha opts for not maintaining the inclusion or subset property. Non-inclusive on-chip cache hierarchies have been previously proposed by Jouppi and Wilton [21] in the context of a single-CPU processor. However, the use of this technique in the context of multiple CPUs 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 copy of the L1 tags and state at the L2 controllers. In this scheme, each controller maintains tag/state information for L1 lines that map to it given the address interleaving. In the following we refer to the L1 state/tag kept in the L2 controllers as dup-L1 state/tag. The total memory overhead for dup-L1 state/tag across all controllers is 64KB, which is less than 1/32 of the total amount of on-chip memory.

In order to lower our 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. Our L2 effectively behaves as a very large victim cache that is filled only when data is replaced from the L1s. This implies that even clean lines that are replaced from an L1 may cause a write-back to the L2. Given that multiple L1s may have copies of the same line, a write-back of the replaced data may not be necessary. To avoid unnecessary write-backs, the dup-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.

Given our virtually indexed L1 caches, we also need a mechanism to deal with the presence of multiple copies of a line in the L1 under different virtual addresses (i.e., synonyms). Unlike most designs that check all possible synonym locations in the L1 in parallel with the L2 lookup, Piranha delegates such checking to the L2. Our checking for synonyms is simplified by the fact that the dup-L1 tag are physically indexed.

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 dup-L1 state/tag and the state/tag of the L2 itself are checked in parallel. Therefore, our intra-chip coherence protocol 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 (or nodes) and if so, whether it is cached exclusively. This partial interpretation allows the L2 controller at home to avoid communicating with the protocol engines for the majority of local L1 requests. This information is also kept
in the L2 and dup-L1 states to distinguish between local/remote chip states. A line is considered to be remote if there are any copies at remote nodes, and is otherwise considered local. Maintaining this condensed information at the L2 allows us to do the optimizations mentioned above without needing to cache the full directory information at the L2 (or alternatively to fetch the directory from memory on every access, which would be prohibitive).

A memory request from an L1 is sent to the appropriate L2 bank (based on the address interleaving). Based 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 address maps to the local memory, i.e., local home). 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. 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. Requests that are forwarded to the home engine also carry a copy of the memory directory, which is updated by the home engine and written back upon completion of the transaction.

The L2 uses a round-robin (or least-recently-loaded) replacement policy whenever an invalid block frame is not available. The policy was chosen both for simplicity of implementation and because more sophisticated policies are unlikely to yield significantly better results. In particular, giving preference to dirty or remote data in the replacement policy could lead to lower performance since this tends to lengthen the lifetime of such data in the cache. Lengthening the lifetime of shared-dirty data has been shown to adversely affect performance by increasing the already high fraction of dirty remote misses in workloads such as OLTP [3].

2.4 Memory Controller
Piranha has a high bandwidth, low latency memory system based on direct Rambus RDRAM. In keeping with the modular design philosophy of the chip, there is one memory controller and associated RDRAM channel for each CPU/cache, for a total of eight memory controllers. Each of the eight Rambus channels can support up to 32 RDRAM chips. In the 64Mb memory chip generation, each Piranha processing chip can support a total of 2GB of physical memory (and 8GB in the following RDRAM generation). Each RDRAM channel has a maximum data rate of 1.6GB/sec, with most of this bandwidth available in a sustained manner. Therefore, a single Piranha processing chip has a maximum local memory bandwidth of 12.8GB/sec. The latency for a random access to memory over the RDRAM channel is 60ns for critical word first, and an additional 30ns for the rest of the cache line.

In contrast to other chip modules, the memory controller does not have a dedicated port to the intra-chip switch. Access to memory is controlled by and routed through the corresponding L2 controller. L2 commands to the memory controller are at the granularity of a cache line. The L2 can issue read/write requests for both data and the directory associated with it. Details on the encoding of the directory are described in Section 2.5.2.

The design of the memory controller consists of two parts: the Rambus ASIC Cell (RAC) (provided by Rambus) and the memory controller engine (specific to Piranha). The RAC incorporates all the high-speed interface circuitry required to access the Rambus channel. 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 channels and 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, the home engine and the remote engine, 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 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 [33]. Figure 4 shows a high-level block diagram of
the protocol engine. The remote and home engines each use an instance of the microcode controlled datapath shown in the figure. Each engine consists 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. This is a special purpose controller, which implements a few powerful microcode instructions that are specifically tailored towards the execution of cache coherence protocols. The microcode memory supports 1024 21-bit-wide instructions. 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 (send to local node), LRECEIVE (receive 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. To determine the successor, a 4-bit condition code is OR-ed into the least significant bits of the 10-bit next-instruction address field.

The actual protocol code is specified at a slightly higher level with symbolic arguments and labels for goto statements, and a microcode assembler is used to do the appropriate translation and mapping for the microcode memory. Typical cache coherence transactions require only a few instructions to complete at each engine that handles the transaction. For example, the typical path for a read transaction to a remote home involves a total of four instructions 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 sends the reply 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.). The TSRF entry remains allocated until the full completion of the transaction. 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.

To achieve microcode instruction issue on every cycle (for 500MHz operation), we need to use an interleaved execution model that alternates between even- and odd-addressed threads on every cycle. A single cycle is not sufficient for computing the address and fetching the next target instruction for a given thread based on the condition codes generated in the current cycle. The interleaving allows us to fetch the next instruction for an even-addressed (odd-addressed) thread while executing the instruction for an odd-addressed (even-addressed) thread.

We believe that the above design provides a nice balance between flexibility (e.g., for late binding of protocol features) and performance. While our design is less flexible than using a general-purpose processor as in FLASH [25], 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. This is achieved by computing ECC at a coarser granularity and utilizing the unused bits for storing the directory information [32,39]. In our design, 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 data path for directories, this approach leads to lower cost by requiring fewer components and pins, and provides simpler system scaling. Furthermore, given the trend towards larger main memories, dedicated directory storage can become a significant cost factor. Finally, 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: one bit indicates an exclusive remote copy, and the second bit distinguishes between limited pointer and coarse vector representations. The directory is not used to maintain information about sharers at the local node, and an all-zero directory indicates there are no remote sharers. Furthermore, directory information is maintained at the granularity of a node and not at the granularity of individual processors. Given the two bits of state, we have 42 bits to encode sharers. Assuming a system with 1K nodes, we can represent up to 4 remote nodes with limited pointers and switch to the coarse vector representation for more sharers.

2.5.3 Inter-node Coherence Protocol
Piranha uses an invalidation-based directory protocol to maintain coherence across nodes. The protocol supports four types of requests: read, read-exclusive, exclusive (if the requesting processor already has a shared copy), and exclusive-without-data. We also support a clean-exclusive optimization whereby a read request is returned an exclusive copy if there are no other sharers. The protocol also supports writeback and replacement requests (the latter is only used when a clean-exclusive copy is replaced). In case of a remote owner, the home node forwards the request to the owner, which in turn directly replies to the requesting node. The protocol also supports eager exclusive replies whereby ownership is given to a processor before all invalidations are complete. Invalidation acknowledgments are always 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 exploiting multiple virtual lanes and depending on sufficient buffering in a few cases (explained in more detail below). We avoid the second use of NAKs by appropriately covering the various transient cases so that forwarded requests are always guaranteed to 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 acknowledges the writeback. This allows the owner node to satisfy any incoming requests that may have been forwarded by the home node before it observes the writeback. 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.4

The lack of NAKs and 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 when a request first reaches the home. This property eliminates the need for extra messages that are sent back to the home to confirm that the forwarded request is satisfied (e.g., “ownership change” messages that are common in scalable protocols such as DASH [27]), and also eliminates the associated protocol engine occupancy to handle such messages. Therefore, our protocol can handle three-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 and retries. For example, the SGI Origin protocol [26] uses a number of complicated mechanisms such as keeping retry counts and reverting to a strict request-reply protocol to avoid such problems, while most other protocols with NAKs simply ignore this important problem (e.g, DASH [27], FLASH [25]).

To avoid deadlock, our protocol uses three virtual lanes (I/O, L, H). The I/O lane is used by I/O requests (explaining the need for this lane is beyond the scope of this paper). The low priority lane (L) is used by requests

---

3 This corresponds to the Alpha write-hint instruction (wh64) which indicates that a processor will write the whole cache line, and therefore does not care about its current contents. This optimization is especially useful in copy routines.

4 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.
sent from a remote node to a home node (except for writeback/replacement requests which use H). Finally, the high priority lane (H) is used by forwarded requests and all replies. Our deadlock solution also relies on sufficient buffering in the network. The overall size of this buffer across the network is proportional to the number of requests that can be injected into the system (bounded by the number of TSRF entries at each node).

We use a number of unique techniques to limit the buffering requirements per node to a reasonable number of entries that does not grow with the number of nodes. First, the network uses “hot potato” routing with increasing age and priority for messages that are not making progress towards their final destination. This enables a message to theoretically reach an empty buffer anywhere in the network. Therefore, the total amount of buffering in the network needs to 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 that can be 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. CMI is only used when the number of nodes to be invalidated exceeds a certain threshold. Furthermore, our studies show that it leads to superior invalidation latencies by avoiding the major serializations that arise from injecting a large number of invalidation messages from the home node and gathering the same number of acknowledgments at the requesting node. These properties allow us to provide a limited amount of buffering per node that does not need to grow as we add more nodes. Sufficient buffering in the network also simplifies the design of the protocol engines since an engine can stall on a thread that is attempting to send an outgoing message without needing to preempt the thread (for deadlock reasons) to handle incoming messages from other lanes.

## 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. The RT also deals with traffic that transits through this node on the way to another node. Transit traffic does not impact any component of a Piranha node other than the RT. The IQ receives packets that are addressed to the local Piranha nodes and forwards them to the appropriate 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, which initiates different actions without any intervention by the CPU cores. These actions include reads and writes of configuration registers, access to the on-chip memory, download of a routing table, and testing of 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 [31]. Like the S-Connect, the RT also 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 different nodes for redundancy. The system interconnect uses only two distinct packet types. The **Short** packet format uses 128 bits and is used for all data-less transactions. The **Long** packet is used to transfer data and uses the same 128-bit header format along with a 64 byte (512 bit) data section. Hence packets are either 128 bits or 640 bits in length and are transferred in either 2 or 10 interconnect clock cycles.

An 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 signalling rate is four times the system clock frequency, or

---

5 As a concrete example of the amount of buffering required per node, 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 TSRF’s * 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!.
2Gbits/second 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 the electrical problems of 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 signalling 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. The code is designed such that the 19th bit is encoded by inverting all 22 bits. Hence the Piranha code is inversion insensitive. This property is used to randomly invert the data on each link so that statistical DC-balance is achieved in the time-domain along each wire\(^6\). As a result, 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 Output Queue (OQ)

The OQ temporarily stores packets before they are accepted by the router for transmission to other nodes. The purpose of the OQ is to provide a modest amount of buffering in order to de-couple the operation of the router from the local node. Naturally, the fall-through time in an uncongested system is critically important in minimizing the latency for access to other nodes. The OQ optimizes this path and imposes only a 1 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 internal 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 entire Piranha interconnect system.

### 2.6.3 The Input Queue (IQ)

The IQ receives packets from the RT and forwards them to their destination modules via the packet switch. It is important to remove terminal packets from the RT as soon as possible because the high 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 and does not allow low priority traffic to block high priority traffic. In order to improve overall system performance, the IQ logic allows low priority traffic to bypass high priority traffic in the case that 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 in order to determine the appropriate 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 will be directed at the protocol engines. Some packets, which are related to initialization, configuration, error handling, interrupt processing, and performance monitoring, 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 data paths, redundant data paths, protocol error recovery\(^7\), 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 to RAS challenges in large-scale systems is beyond the scope of the project, our design provides hardware hooks to enable future research in this area. These hooks come in the form of well-defined transactional semantics both in the intra-chip switch and router interfaces and in the limited flexibility available through the use of our 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

---

\(^6\) It is possible to do better than random inversion, however the necessary logic would require an extra pipeline stage.

\(^7\) In Piranha, the TSRF in the protocol captures the state within the cache coherence protocol and keeps track of outstanding messages. Based on the TSRF state, a protocol engine can monitor for failures via such mechanisms as time-outs and error messages. If the protocol engine cannot handle the error, the TSFR entry can be encapsulated in a control message and directed to recovery or diagnostic software.
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 System Software Strategy

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. We are expecting a minimal porting effort for the OS (Tru64 Unix), with changes expected only at the level of some device drivers and some tweaking of the affinity parameters to reflect the system characteristics of Piranha.

Our biggest software effort will be in porting the Alpha console, which is a separate piece of software from the OS that is traditionally used in Alpha systems for bootstrap and device discovery. Finally, we expect a minimal effort for developing the PAL code for Piranha. This is because we are implementing the same PAL instructions and internal processor register formats as the Alpha 21364, with a few minor deviations.

OS scalability was an early concern because Piranha replaces a single CPU with eight less powerful cores on each chip. However, as our results show in Section 5, the scaling properties for Piranha are quite good.

4 Evaluation Methodology

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

4.1 Workloads

Our OLTP workload is modeled after the TPC-B benchmark [44]. 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 [45]. 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].

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 which validated such scaling [4]. We use a TPC-B database with 40 branches with a shared-memory segment (SGA) size of over 900MB (the size of the metadata area is over 100MB). 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.

4.2 Simulation Environment

For our simulations, we use the SimOS-Alpha environment (the Alpha port of SimOS [38]), 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.

### 4.3 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.

### 5 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 approach. We use the OLTP and DSS database workloads described in the previous section for this evaluation.8
Figure 5 shows our results for single-chip configurations for both OLTP and DSS. We study three 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 differences between OOO and P8. The figure shows execution time normalized to that of OOO. The execution time is further 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 almost 2.5 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 1.7 times, with wider-issue and out-of-order providing the remaining 1.45 times gain. However, once we integrate eight CPU’s, the single-chip Piranha (P8) outperforms OOO by almost 2.9 times.

Several factors contribute to Piranha’s exceptional performance on OLTP. Figures 6(a) and (b) provide more detailed data to better explain this. Figure 6(a) shows speedup as we increase the number of on-chip CPUs in Piranha, while Figure 6(b) shows the behavior of the L2 cache. The latter figure shows a breakdown of the total number of L1 misses (normalized to P1) that are served by the L2 (L2 Hit), forwarded to another on-chip L1 (L2 Fwd), or served by the memory (L2 Miss). As shown in Figure 6(a), eight CPU’s achieve nearly a 7 times speedup over a single CPU. This speedup arises from the abundance of thread-level parallelism in OLTP, along with the extremely tight-coupling of the on-chip CPU’s (leading to small communication latencies) and the effectiveness of the on-chip caches in Piranha. The last effect is clearly shown in Figure 6(b). Although the fraction of L2 hits drops from about 90% to under 40% when we go from 1 to 8 CPU’s, the fraction of L2 misses that go to memory remains constant at under 25% past a single CPU. In fact, adding CPU’s (and their corresponding L1s) in Piranha’s non-inclusive cache hierarchy actually increases the amount of on-chip memory (P8 nearly doubles the on-chip memory compared to P1), which counteracts the increased pressure on the L2. The overall trend is that as the number of CPUs increases, more L2 misses are served by other L1’s 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). These results indicate that Piranha’s non-inclusion policy is extremely effective in utilizing the total amount of on-chip cache memory (i.e., both L1 and L2).

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, Piranha 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). Second, since OLTP is mainly latency bound, the eight CPUs still do not come close to saturating on-chip or off-chip bandwidth. Third, OLTP workloads have been shown to exhibit constructive interference in the instruction and data streams [28].

Referring back to Figure 5, we see that Piranha (P8) also outperforms OOO for DSS, although by a narrower margin than for OLTP (2.3x). The main reason for narrower margin comes from the workload’s smaller memory

---

8 We have also done an evaluation of Piranha with automatically parallelized SPECfp95. Our early results showed reasonably good performance, with an 8-CPU single-chip Piranha being comparable to an 8-processor 600MHz 21164 system. We plan to re-run these experiments to reflect more recent changes to our design and will include them in the final version.

9 This is consistent with results reported by Ranganathan et al. [36] which also show limited gains from wide-issue and out-of-order execution due to the lack of instruction-level parallelism in OLTP workloads.
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 which 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 use of 8 CPU’s in Piranha (P8) leads to an almost perfect speedup of 8 relative to a single CPU (P1).

One interesting alternative to consider for Piranha is to trade CPU’s for a larger L2 cache. However, since the fraction of L2 miss stall time is relatively small (e.g., 25% for P8 in Figure 5), the improvement in execution time from even an infinite L2 would also be small. Moreover, since Piranha CPU’s 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 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\(^\text{10}\) (i.e., P4). The figure shows that the Piranha system scales better than OOO (3.1 vs. 2.3) 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

---

\(^{10}\)The current version of the operating system that we use in our simulation environment limits us to 16 CPUs. The final version will include these scalability numbers for a fully configured Piranha chip (i.e., 8 CPUs per chip for a total of 32 CPUs).
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 used to implement the prototype. To fairly judge the potential of the Piranha approach, we also evaluate the performance of a full-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 4.8 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 benefits more from the 150% boost in clock speed (P8 vs. P8F). The gains in OLTP also are 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.

6 Design Methodology and Implementation Status

Our design methodology starts with C++ based models which serve as the official specification of each of the major Piranha modules (e.g. L2 caches, memory controller, etc.). The C++ models implement the behavior of the modules in a cycle-accurate fashion and use the same boundary signals as in the actual implementation. These models are then used as 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. For the other Piranha modules, we are in the process of completing cycle-accurate behavioral models with verilog to follow. The C++ models are much faster than their Verilog counterparts, allowing lots of runs for functional and architectural verification. Our environment also allows for 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 [20]. This design flow incorporates 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 \( \mu \text{m}^2 \) [6] and gate delays of 81ps (worst case) for an unloaded 2-input NAND.

Although we have not completed the entire Piranha implementation, we can infer the clock frequency from preliminary logic synthesis of the processor core and comparisons to comparable microprocessor cores in similar technologies. We have also estimated 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 processor node illustrated in Figure 9. The eight processor/memory slices are clearly visible in the floor-plan, each consisting of an Alpha processor core with L1 caches, a 128 KB bank of L2 cache, along with a memory controller and RDRAM RAC cell. Roughly 75% of the Piranha processor node area is
7 Discussion and Related Work

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

There have been a large number of recent studies of database applications (both OLTP and DSS) due to the increasing importance of these workloads \([3,4,7,8,12,22,28,29,35,36,37,43,46]\). To the best of our knowledge, this is the first paper that provides a detailed evaluation of commercial database workloads in the context of chip multiprocessing. Ranganathan et al. \([36]\) study simulations of 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. These results are consistent with our results presented in Section 5 which are based on full-system simulations including kernel activity. A number of studies address issues related to the size and effectiveness of second-level caches for OLTP workloads. Barroso et al. \([4]\) show the need for large direct-mapped off-chip caches (8 MB). Lo et al. \([28]\) 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 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) especially when coupled with our technique of extending the amount of available on-chip memory by not maintaining inclusion in the L2 cache. Finally, Barroso et al. \([3]\) show that aggressive chip-level integration of memory system, coherence, and network modules on a single chip (as in Alpha 21364) can provide large gains for OLTP workloads. Piranha exploits such integration along with including multiple processor cores on a single chip.

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. An anecdote related to the unveiling of the Control Data 6600 in August of 1963 is instructive \([19]\). Thomas Watson (then CEO of IBM) is quoted as saying: “Last week Control Data ... announced the 6600 system. I understand that in the laboratory developing the system there are only 34 people including the janitor. Of these, 14 are engineers and 4 are programmers ... Contrasting this modest effort with our vast development activity, I fail to understand why we have lost our industry leadership position by letting someone else offer the world’s most powerful computer.” The response from Seymour Cray was: “It seems like Mr. Watson has answered his own question.” Indeed, the CDC6600 had a narrow target market (scientific computing), and hence the machine could be specialized for that market. This historical lesson has not gone unheeded given a number of contemporary processor designs that are specifically focused on commercial markets \([5,24]\).
Several papers from Stanford have advocated and evaluated the use of chip multiprocessing (CMP) in the context of workloads such as SPEC [15,30,34]. There are currently a number of implementations in progress that are exploiting CMP. The Stanford Hydra project is exploring CMP with a focus on thread-level speculation [16,17]. The current implementation is integrating 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 uses 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 functionality 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 [23,42] is not necessary for achieving high performance on such workloads and can in fact lead to non-optimal design trade-offs.

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. Even though the cores each have their own 16KB instruction cache, they 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) (along with other forms of multithreading) is an alternative to CMP for exploiting the thread-level parallelism in commercial workloads. In fact, Lo et al. [28] 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 carries over the implementation and verification complexity that comes with such designs. Furthermore, SMT performance is influenced by the fact that critical resources, such as the physical register file, L1 caches, and TLB’s, are shared simultaneously by multiple threads. To avoid negative effects due to this sharing, software must in some cases use intelligent resource management (e.g., in the form of page-mapping and data layout policies) [28]. The advantage of SMT over CMP is that it provides superior performance on workloads that do not exhibit thread-level parallelism because it has a powerful underlying processor core. 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, have similar behavior to decision support (DSS) workloads [4]. We hope to evaluate a broader range of web server applications in the near future.

8 Concluding Remarks

The use of chip multiprocessors 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 complexity issues. 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 question for designers of future processors will not be whether to use chip multipro-
cessing, but how powerful each processor core should be (which determines the number of cores that can be integrated onto a single chip) 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 2.8 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. 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. 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, which have the potential of providing superior performance on commercial workloads while addressing some of the complexity issues that have plagued recent processor designs. 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 the 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 Verilog of the Alpha core, Dan Scales helped with the inter-chip coherence protocol, Basem Nayfeh was an early member of the architecture team, Robert Bosh 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. Finally, we are grateful to Jeff Mogul and Keith Farkas for their careful review of the manuscript.

References


