In Proceedings of the 30th International Symposium on Computer Architecture (ISCA), June 2003.

A Pipelined Memory Architecture for High Throughput Network Processors

Timothy Sherwood George Varghese Brad Calder

Department of Computer Science and Engineering
University of California, San Diego
{sherwood,varghese,calder}@cs.ucsd.edu

Abstract

Designing ASICs for each new generation of backbone routers is a time intensive and fiscally draining process. In this paper we focus on the design of a programmable architecture for backbone routers, based on the manipulation of wide irregular memory words, that can provide a feasible design alternative to custom ASICs. We propose a pipelined memory design that emphasizes worst-case throughput over latency, and co-explore architectural tradeoffs with the design of several important network algorithms. Through this co-exploration, we show that a programmable architecture can efficiently exploit behavior inherent to most common network algorithms to keep up with next generation network speeds.

1 Introduction

The constant march of fiber optic link speeds has in recent years turned into a full speed sprint that has out paced even the exponential growth predicted for integrated circuits by Moore's law. Building backbone routers capable of handling these speeds currently requires an army of design and verification engineers, large amounts of silicon, and years of design time.

In order to handle a single 10 Gigabit per second fiber link (and many routers have 32 links), each line card must process up to 32 million packets every second[1]. Next generation products will have to handle upwards of 125 million packets per second. Even as bandwidth requirements are increasing, so is the complexity of protocols that need to be supported — from traditional forwarding functions such as IP lookup, to more recent requirements for doing intrusion detection.

Further, a competitive backbone router must not only support the most recent protocols, but must also take advantage of the most recent advances in network algorithm design. This is in stark contrast to more traditional architecture research where it is assumed that the benchmark is an unchanging standard. In the core router setting the *only* requirement is bandwidth, and anything in the design space can change to maximize bandwidth, including the architecture and the algorithms themselves.

In this paper we describe a programmable system designed to efficiently execute a variety of network algorithms and show that it is a feasible alternative to custom ASICs. To achieve the high throughput required by core routers, we propose a novel memory design for our network processor and make use of a form of non-uniform wide word parallelism.

[1] assuming a minimum packet size of 40 bytes

We begin the paper in Section 2 explaining the design requirements of modern throughput-driven backbone routers and show how they differ from traditional latency-driven computing architectures. The unique requirements of processing network algorithms requires a new bandwidth-centric architecture that can deliver optimized worst-case performance. In Section 3 we summarize, at a very high level, the space of network processors and discuss how our design fits into this space. In Section 4, we then describe and characterize our high throughput memory design, and contrast it with memory designs geared towards minimal latency. A key point of this research is the importance of understanding both architectural and algorithmic constraints at the same time. To this end, Section 5 focuses on co-exploring tradeoffs in both the architectural and algorithmic dimensions, and shows how a single architecture can perform well across a broad set of applications, if the algorithms are tuned in unison with the architecture. The paper is then concluded in Section 6 with a summary of our contributions.

2 Routing in the Backbone

The goal of our research is to develop an architecture capable of implementing network algorithms at the speeds required by future backbone routers (10 Gb/s to 40 Gb/s), while remaining general purpose enough that new algorithms can be implemented completely in software or with a minimum of additional hardware complexity. In contrast, most programmable network processors on the market today target low performance (from 100 Mb/s to 10 Gb/s) low cost edge routers, leaving the task of routing in the backbone to ASICs.

2.1 Backbone Router Model

The following model of a how a packet and router interact will help to ground the discussion that follows. A packet arrives at a router on an input link to an input line card that contains a router processor. This processor first does any security checks (e.g., screening for suspicious strings) and then performs a lookup. The lookup maps the destination address to the router output link that the packet should be sent to. The packet is then switched, using an internal crossbar, to the output line card specified by the matched prefix.

The input processor may additionally classify the packet using a database of rules on header fields. At the output line card, classified packets are buffered in separate queues for each traffic class (e.g., low delay, high priority). The output line card also contains a processor which schedules packet queues for transmission on the output link in order to provide QoS (e.g., delay)

guarantees.

The traditional approach to high speed router forwarding is to build custom ASICs that perform the required functions at these high speeds. Designing such ASICs is increasingly becoming a Herculean effort. As bandwidth requirements grow, network protocols become more complicated, and higher frequencies complicate physical design. Each new generation of routing ASIC is larger and more complex than the last, requiring larger teams from design to verification.

To combat this increasing complexity, a software based approach is needed that requires fewer engineers and less time. For software to run at the speeds required, we must create a new class of efficient, easily programmable, and scalable architectures to which we can map many different network algorithms.

A generalized routing architecture has two major advantages over the traditional method that uses a myriad of ASICs. First, a major concern for router designers is to reduce Non-Recurring Engineering (NRE) costs (e.g., for masks, design time, etc.) which are becoming a dominant fraction of the overall costs for ASICs as speeds increase. Second, a programmable processor allows changes to the algorithms after installation. Such changes were not required for the traditional IP path that was cast in stone, but is becoming increasingly necessary today for security applications.

To address these issues, we focus on the design of a programmable architecture capable of handling modern network algorithms at backbone speeds. Note that our goal is *not* to replace the entire router with programmable parts. We continue to assume a hardwired internal crossbar switch, custom logic for buffer handling, etc.; the programmable processor only implements complex algorithms with large amounts of state such as IP lookups, classification, scheduling, and security checks.

2.2 Routing Requirements

Many initial attempts at designing network processors essentially retrofitted general purpose computer architectures to work in the networking space. While these are acceptable at low speeds, the goals of a programmable router processor are very different from a general purpose computer, and as such are governed by a different set of requirements. This leads to the following key differences in design assumptions that we address in our design:

- *Throughput is critical, not latency:* A single 40 Gigabit link, forwarding minimum size packets (40 bytes), requires that 125 million packets be handled each second (one every 8 ns). The handling of each packet requires multiple memory references, causing memory bandwidth to be of paramount importance. However; while the demand on throughput is very high, each individual packet may take many hundreds of nanoseconds from arrival to departure.
- *Design for the worst case:* This requirement is contrary to the current trends in processor designs that are based on optimizing for the average or expected case. In contrast, network vendors rate their products as supporting a particular wire rate. It is important that the performance delivered by their product be stable for any combination of test packets with which a customer chooses to test it. [2] This greatly complicates the design and analysis of a practical memory system.
- *Do not assume locality in the packet stream:* The final important point of difference follows from the previous two. Exploiting various forms of locality (e.g., spatial, temporal) in data sets is a tenet of computer architecture, one that has made possible the common case performance of desktop systems today. Unfortunately, because backbone links merge together millions of individual packet streams [25], and because our goal is worst case throughput, we cannot rely on inter-packet locality to improve performance.

The standard worst case assumptions used by network designers [15] is that packets are minimum sized (roughly half the packets on the Internet are 40 byte TCP acknowledgments [25]), and there is no locality between packets.

2.3 Our Approach

Our approach to designing a 40 Gb/s router processor was to first understand the algorithms and techniques in use on many of today's high end router ASICs. We generalize from these examples to propose a programmable architecture capable of achieving high bandwidth for worst case performance. For each algorithm, we build an analytical model of its behavior and characterize the performance of the algorithm working in unison with the architecture. We use these models to explore design tradeoffs between the efficiency of the algorithm, and the circuit design options available for implementation. In our analysis and simulations we always assume a worst case packet stream for each algorithm. Likewise, all the results we present are lower bounds on the achievable performance across all possible packet streams.

3 Network Processors

In this section we describe two popular network processor models (shared memory multi-threaded multi-processors, and pipelines of processors using partitioned memory). We then describe our underlying wide-word processing model, made possible by deeply pipelining on-chip memory. As with any interdisciplinary work, we can only touch upon a sampling of related work in network processors and refer the interested reader to [22] for a more comprehensive survey.

3.1 Independent Processors with Shared Memory

The top of Figure 1 (part a) shows a common architecture for many network processors such as the Intel IXP1200. Internally, the processor contains several (e.g., 6 for the IXP) CPUs. The packet stream is divided among CPUs, with each processor working on multiple separate packets (threads) at the same time. When the processing for one packet stalls because of a memory reference, a CPU resumes processing for the next thread. Using fast context switching between threads, and 4 contexts per processor, the IXP1200 can process 21.4 M packets/second [23].

The main drawback of this design is that it is built around traditional memory, which is optimized for latency rather than

[2] A router that cannot process packets at wire rates can drop (possibly important) packets before they are even examined and additionally may be susceptible to Denial-of-Service attacks using worst-case traffic.

throughput. To achieve high memory bandwidth, multiple processors and thread contexts are used to generate multiple pending accesses. The problem is that each processor does all of the forwarding for an assigned packet, and each independent processor needs independent access to a large amount of shared state (e.g., the global forwarding database.).

The only known way around this problem is to either replicate the state for each processor (e.g., replicate the 100,000 entry forwarding tables across all CPUs), or to share the state in some fashion. Replication is prohibitively expensive for the large problem sizes that routers need to deal with, especially given limited (e.g., 32 Mbits of SRAM) on-chip memory. The problem with sharing state is in providing *independent* access to the entirety of the state. In the worst case, if every data access of every processor was to the same bank of memory, all the benefit of multiprocessing would be lost because all memory accesses would be serialized. While these techniques work well on systems with small problem size or without worst-case performance constraints, neither of these options meets the requirements we set forth in Section 2.2.

3.2 Pipeline of Processors

The lack of memory bandwidth in the traditional shared memory model has led some companies like Intel and AMCC to move to a pipeline of processors, with multiple distributed memories. The model is that each processing element in the pipeline does part of the processing of a packet for *all* packets; for example, in the middle of Figure 1 (b) processing element 1 can do the initial processing of a lookup and pass the results to processing element 2 for completion. The drawbacks of pipelined architectures are:

a) Memory Utilization/Contention Tradeoff: The simplest way to design the pipeline is to statically partition the processing state (e.g., IP lookup table) among processors to avoid contention, as shown in Figure 1 (b). In algorithms such as IP lookup, the data structure is a tree, and the different levels of the tree are partitioned among the pipeline of processors. The work at the root of the tree is assigned to the first processing element, the root's children are assigned to the second element, and so on. Unfortunately, the trees are not balanced; the shape of the tree can vary from lookup database to database. Thus the memory needs of each CPU in the pipeline becomes unpredictable. This in turn means that static partitioning of the limited available on-chip memory among multiple distributed memories can be very wasteful of memory.

b) No support for writes: A pipeline of processors work best for functions such as lookups where writes are done only occasionally and not in real-time. However, there are important router forwarding functions that require forwarding time writes. For example QoS scheduling involves real-time writes to update flow tracking registers, which then may have to be forwarded to all of the other independent memories. This make a worst case analysis on the mapped algorithm very difficult.

3.3 Our Solution: Pipelined Wide Word Memory

Our solution is motivated by the problems described above, which are all caused by the need for worst case memory bandwidth. We avoid these problems as follows. First, we make each memory access wide to increase the number of bits retrieved per access. Second, we observe that a memory subsystem consists

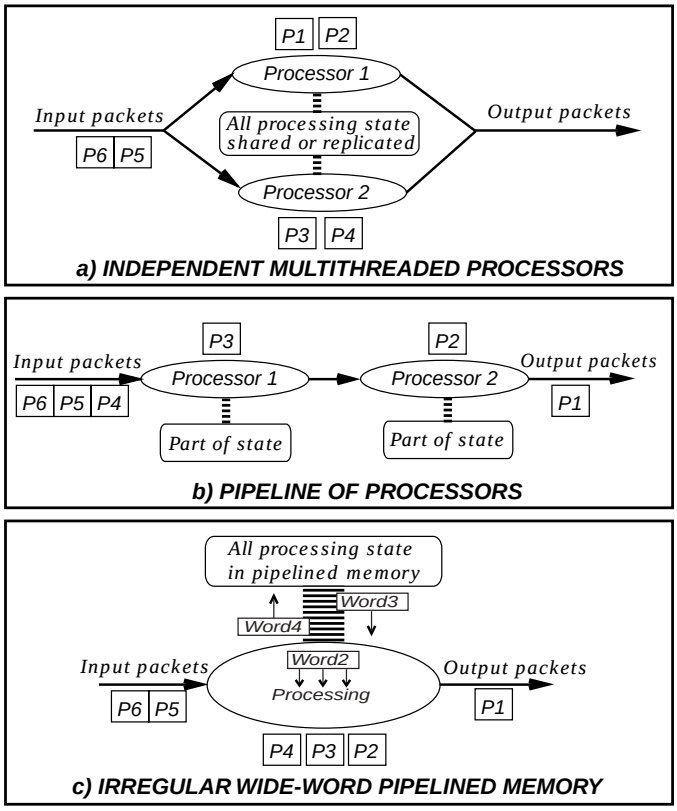

Figure 1: Two common approaches to parallelism in network processors versus our pipelined memory design. In the top picture (a), input packets are distributed among multiple processing units to divide the load. Each processing element works on the complete handling of a packet but can work on multiple packets (threads) at a time. Thus each processing element must access all of the memory state. In the middle picture (b), input packets flow through a pipeline of processing elements. Each processing element performs a portion of the processing for each packet, one packet at a time. Thus each processing element need only access separate parts of the memory state. In our scheme (c), there is no need to partition the data structures, and memory throughput requirements are met by using wide words and pipelined memory.

of memory banks and an interconnect that ferries data read from a bank to the output. Traditional memory maintains the invariant that two readers do not access the same block by ensuring that *there is only one reader in the entire subsystem*. However, this is unnecessarily restrictive. In summary, our memory subsystem uses:

- *Wide Words*: We note that the performance of many network algorithms can be increased significantly by the use of wide data words. Instead of having to worry about multiple independent readers of memory having bank conflicts, we can instead provide aggressive memory bandwidth to one reader at a time through wide word access.
- *Internally Pipelined Memory:* Circuit design realities force traditional memories to sacrifice worst case throughput in order to reduce latency. Because our applications are incredibly

latency tolerant, we can greatly increase the throughput by *internally* pipelining the memory to allow the memory system to work on multiple packets at once. For example, in the packet stream of Figure 1, the memory can be delivering a word required to process packet $P1$, while at the same time fulfilling a read request for packet $P2$.

To best utilize our wide-word pipelined memory, our underlying processor model is also different from either of the architectures described above (shown in Figure 1 (a) and (b)). Our design in Figure 1 (c) has one multi-threaded processor working on multiple packets at the same time. When a wide word is delivered from the memory system, multiple computation units work together over the whole word performing the calculations needed to process that word and generate the next address. Multiple packets have their processing interleaved, as our pipelined memory allows multiple memory requests to be outstanding at a single time.

4 Memory Architecture

In this section we describe the details of our approach, including a description of the tradeoffs of wide word and pipelined memories, the model of the machine presented to the programmer, and a characterization of the architecture.

4.1 Motivation for Wide Word Memory Architectures

Network algorithms traverse large data structures and are designed to provide worst-case bounds on both processing time and memory. The use of wide memory word accesses is very common within network ASIC designs as a means of increasing memory bandwidth and reducing processing times. An example of this can be found in the way that tree-based IP lookup algorithms are typically implemented. IP lookup can be efficiently implemented by storing prefixes in a *trie* that looks up one bit of the address at a time and goes either right or left through the trie based on this bit. In hardware trie algorithms, a state machine traverses the trie using bits in the destination address to obtain the output port. One technique to decrease search time is to reduce the tree height by packing multiple nodes in a trie path into a single wide word [24]. This allows more than one bit to be looked up at a time using a single memory access, but increases the size of the memory access and the fan-out of the trie. This is just one example of how wide-words can be used in network algorithms. In Section 5, we briefly explain for each of our algorithms how a wide word can be used to speed up the algorithm.

Because of the common use of wide words in a number of special purpose networking ASICs for lookups, classification and QoS, we believe it is important to abstract this ability in a programmable architecture by exploiting wide word access in the memory design and exposing wide word access at the programming level.

4.2 A High Throughput Pipelined Memory

While multithreading and pipelining are very standard ideas in the network processor context, one of our main contributions is showing how to get high worst case memory throughput by internally pipelining the memory. In the traditional memory layouts that prior network processors are designed around, only one or two (if multi-ported) memory lookups can be performed at the same time in the worst case. These designs use large memory tiles, trying to minimize the interconnect delay for latency, and rely on common case techniques such as locality and independent accesses to provide sufficient bandwidth. As such, the worst case throughput of such systems is limited to one or two transactions being processed by the memory at the same time, when, in the worst case, all the lookups fall into the same bank.

In contrast, our new memory design trades off latency for throughput. We propose pipelining the memory design using smaller memory tiles to process many transactions at the same time (see Figure 2). This will result in a higher latency per transaction, but the number of transactions per unit time will be significantly higher than in a traditional memory design. To achieve this, we start by breaking the memory into smaller tiles, each of which is capable of reading out a wide word each memory cy-

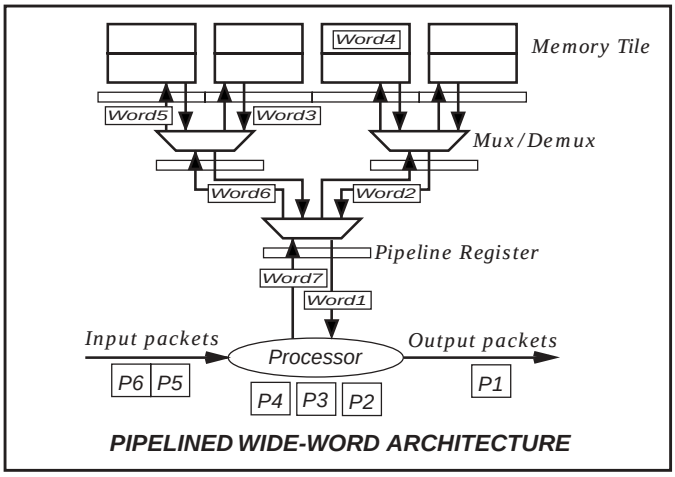

Figure 2: This figure shows how our pipelined tree-based memory can have many accesses flowing through it at the same time. At any given time, there is only one wide word access going to a memory bank, and all the memory banks are at the leaves of the tree. At each level in the tree (for the single ported version) there are at most two words in flight, one going up to the memory arrays and one coming back down.

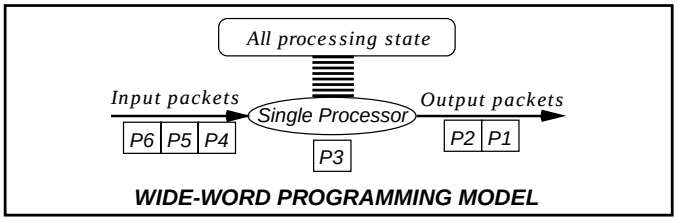

Figure 3: While the actual architecture shown in Figure 2 is quite complex, the programmer need only worry about programming their algorithms for a simple wide-word processor. This is because the architecture is designed to provide good performance even in the worst case, so there is no need to worry about the careful location of data structures in banks, or partitioning the program state between two separate memories as in Figure 1(b).

cle. The cycle times for these smaller banks are two to three times faster than monolithic memory banks, even assuming optimal internal partitioning. Our design will have a larger interconnect delay, and hence a larger overall latency; however by pipelining the interconnect, we achieve better overall throughput because the delay of each stage is smaller than the overall latency of a traditional (non-pipelined) memory design. Therefore, the memory design can have N lookups in flight, where N is the number of pipeline stages (twice the height of the tree to go up and back down) of the memory layout.

Figure 2 shows a high level diagram of our design. The memory has been divided into many small banks. At the bottom of each small bank is a pipeline latch that grabs the data we need. Data read from a bank then needs to travel from the tile (which could be quite far away) back to the processor. This is done over a deeply pipelined tree interconnect. When an access is made, a lookup is performed in parallel at the leaves of the tree, and each cycle the correct data item marches its way down through the pipelined tree to the root where it is read out by the processor. Since the tree is balanced and full, and there is only one memory access actively accessing the banks at a time, we are guaranteed that during the entire journey over the interconnect there will be no conflicts. This ensures that we will always be getting one wide word of data each cycle. The tree interconnect is not a trivial piece of real estate, because it needs to mux, demux, latch, and move large words of memory over a long distance. In fact, as we will see in Section 5, the interconnection network accounts for between 10% and 30% of the total area of the memory system.

By multithreading applications, using small tiled memories with access to wide words, and including an efficient balanced interconnect with worst case bounds, our design effectively supports many in-flight accesses to memory to provide high throughput.

One final hardware optimization which we include in our memory design trade off analysis, is the use of multi-ported memories. Multi-ported memory is slightly slower (Virage reports 10%), significantly less dense (75% larger for the same number of bits), and requires even more overhead in the form of interconnect area. Despite all of these draw backs, multi-ported memory provides two accesses to each memory bank even in the worst case. As we will see later, even though the penalty is significant, using more ports can be a good idea for more aggressive implementations.

4.3 Characterization of Architecture

Now that we have presented the main architectural ideas, we need a method of exploring and testing our ideas along with some intuition as to how the architecture reacts to different parameters.

In order to do this we need a way of quantifying the effect of memory bank delays and areas along with interconnect overheads. To calculate the memory bank parameters we used an analytical model together with a design walker that can estimate the area and delay of .13u SRAM for each bank. This was then validated against industry memory compilers (from Virage and Austria). For the interconnect, we modeled an H-Tree with additional area for pipeline registers and mux/demux modeled from

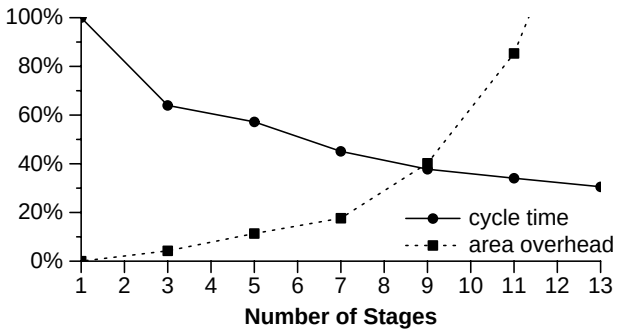

Figure 4: Quantifying the overhead and benefit of pipeline stages. For the line labeled `cycle time`*, the y-axis is the percentage reduction in cycle time for access to a single memory bank. For the line labeled* `area overhead` *the y-axis is the percentage area overhead due to additional interconnect. Remember that the pipeline is not linear, but is instead a tree: this is why the pipeline area overhead increases non-linearly with number of stages.*

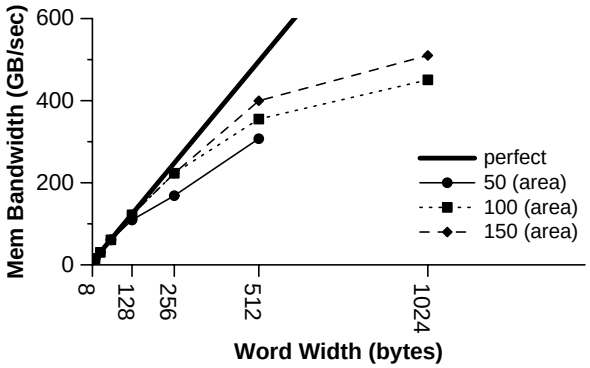

Figure 5: Total memory bandwidth in Gigabytes per second versus the size of word accessed. The line labeled `perfect` *assumes linear scaling of bandwidth with word size. In reality there are diminishing gains above words of 256 bytes caused by interconnect overhead cutting in on the the amount of area we can use for sub-dividing the memory banks.*

VLSI layouts.

To gain some initial intuition as to how some of the key parameters of our memory design trade off against one another, we present Figures 4 and 5. These figures show the effect of increasing the number of pipeline stages and the width of the word used.

Figure 4 shows the overhead and benefit resulting from using different numbers of pipeline stages for a word size of 128 bytes and 1 read-write port. The first line (drawn with a solid line) is the cycle time normalized to a traditional memory organization (which has been banked internally for latency). With only one stage, the memory is not pipelined at all and thus the cycle time is 100%. As we increase the number of stages, the cycle time decreases as the memory is divided into smaller and smaller independent tiles. Doing this sub-division comes at a cost, and the second line quantifies this cost in terms of area overhead (in terms of percent increase in area over a single memory block). The overhead comes from both the cost of splitting a tile into 2

smaller tiles and the increased area due to interconnect.

Figure 5 shows the total memory bandwidth versus the size of word accessed. If there are no circuit level limitations to increasing the size of memory words, the bandwidth should increase linearly (drawn as the `perfect` line in the graph). In reality there is an overhead in increasing the width of the words due to increased interconnect area and slower bank access times. In Figure 5 there are 3 lines drawn, for different constraints on the area. The lines show the maximum memory bandwidth that can be achieved by our system under the given area constraints. Even up to quite large widths (over 128 bytes) there is not that much impact on bandwidth because it is all pipelined. We start to see smaller benefits (caused by interconnect overhead cutting in on the amount of area we can use for sub-dividing the memory banks, and by poor memory folding) when we get to widths above 256 bytes.

The above graphs are just a sampling of the tradeoffs involved in our architecture and do not even include any tradeoffs at the algorithmic level. In Section 5, we provide a detailed examination of the tradeoffs we considered.

4.4 Programmer's View of Memory

We would like to *avoid* forcing the programmer to deal with wide word memory layout problems. For instance, lower speed network algorithms are often implemented using interleaved DRAM chips such as RAMBUS, where great care must be taken by the programmer to lay out the data structure in appropriate banks to avoid bank conflicts. This clearly increases programming effort and in some cases makes efficient design impossible. The task of layout is further complicated by the fact that most of the algorithms are both dynamic and irregular in nature. This, as discussed in Section 3, makes partitioning and replication a very difficult problem to solve, especially in the worst case.

Instead, the model which we would like to present to the programmer is that of a large single unified on-chip memory. It will be our task to make sure that the internal memory design details required for a high speed design are hidden from the programmer. Figure 3 is a high level view of the model we present to the programmer. The figure is simple because the model presented is simple. The memory appears as a large uniform block of wide words. The processor performs a read, does a wide word operation, and then writes that word back to the memory (if it needs to). The challenge is to provide a high performance (high throughput) architecture that can still be programmed in this straight forward way.

4.5 Mapping an Algorithm to Wide Words

While the details of designing and programming algorithms for our architecture are outside the scope of this paper, we comment briefly on how this can be done. The first task is the transformation of the applications to take advantage of the wide-word accesses made available to the programmer. As mentioned earlier, our applications walk large data structures, and thus by default contain many dependent memory accesses. One way that this maps to our architecture is by taking multiple dependent memory references (such as a region of a tree) and flattening them into a single wide word access. However, different algorithms can use the wide words in different ways.

The next task is the processing of each wide word that comes back from memory using a rule based system. The wide word is partitioned into pieces (not necessarily at byte boundaries), and each piece has a small amount of computation performed on it in lock-step (similar to functional units for VLIW). This is followed by a merging step which takes the piece-wise results and makes final decisions or computations (such as which of many next-pointers to follow in a tree).

4.6 Related Architectures

Now that our architecture has been described at a high level, it is important to describe how our architecture differs from other well known architectures which are not specific to networking.

The Imagine stream processor [19] is a close relative to our design. Both are designed for memory bandwidth and focus on total throughput as a metric of performance. However, the Imagine stream processor is built for graphics and signal processing applications that are compute intensive. The Imagine model of computation is one of streaming data between dependent operations, and is built around the reuse and locality commonly found in applications in that domain. The applications we target, on the other hand, navigate through large data structures optimized for worst case performance. Thus Imagine uses reuse to tolerate large memory read times with standard memory designs, while we use a pipelined non-traditional memory to get large throughput without reuse assumptions. On the other hand, our architecture has a higher overall access latency which would be undesirable in many other contexts.

We also share similarities with vector architectures. Vector architectures, much as we propose, operate on very long words of data, or vectors. IRAM [14] is an example of such an on-chip vector architecture. Again, the difference is both in the assumed reuse of a vector and in the applications. Vector machines were built with large regular scientific codes in mind. In order to perform a vector operation the programmer must write his application in the form of vectors. The problem with the network algorithms that we target is that the applications are not regular. Each word that we operate on describes a node in a tree (or a state in a state machine) rather than an array of numbers on which to perform arithmetic.

The final point of comparison is with VLIW architectures. VLIW architectures operate on long irregular *instruction* words. Each word is a set of independent instructions that can operate on any machine state. Our approach is more akin to a *Very Long Data Word* architecture in that we limit operations to the scope of one word at a time. However, our ideas can be combined with a very large instruction word that feeds the logic that works on long data words.

5 Co-Exploring Network Algorithms and Memory Design

The computer architecture community tends towards a passive view of the software that it is tasked with executing. This is because it is simply not possible to make sweeping changes to the algorithms and data structures used in most general purpose environments. By contrast, we propose to take advantage of parameterized network algorithms and data structures that can ex-

ploit our wide word architecture. We do so to expose tradeoffs between the algorithm's need for resources, and the architectural and circuit level realities. This is an important contribution of our work, and in this section we show that co-exploration in this domain space is both needed and useful.

5.1 Algorithm and Memory Design Co-Exploration

In Section 4.3 we examined some of the behaviors of our proposed architecture to illustrate the nature of the tradeoffs we have to deal with, but to do so we made many simplifications in terms of the parameters examined. As we are about to show, in reality the interaction between the parameters is far more complex.

Figure 6 is a graphical representation of the interaction of the parameters in both the algorithmic and architectural design spaces. Software elements are shown as ellipses and hardware elements are shown as boxes. The goal is to fix one of the two outputs (network throughput or area), and then to minimize the other. This graphical representation shows what makes this problem both difficult and interesting: there are many dependencies that interact in complex ways.

For example, if we fix the amount of area allowed, and we want to increase the amount of network bandwidth, we can try to reduce the total number of memory accesses required to process each packet. This can be done by extracting more wide-word parallelism from the application, but in order to do this we need to increase the size of the words read from memory. Increasing the word size affects almost everything else in the system including memory cycle times and interconnect overhead. This example also demonstrates another key point: algorithms are closely coupled with the choice of hardware parameters. Ignoring one, while attempting to optimize the other will be far from the best design.

The results in this section were generated by examining many different architectural and parameter sets, including but not limited to: the width of word in the memory and algorithm, the total size of the memory, the tiling of the memory, the number of read, write, and the read-write ports in the memory. Once the architecture model has been specified, the algorithm is modeled to take advantage of the architecture at hand. Each algorithm takes advantage of the memory width available, the number of ports (read and write) for access, and is limited by the amount of memory available.

While we are certainly not the first to propose design space exploration for network processors [7, 26] we believe that we are the first to systematically co-explore algorithmic and architectural tradeoffs together under worst case performance constraints.

5.2 Network Algorithms

Note that there are two different time scales at which routers process information: *forwarding time* and *route time*. The computations that occur in route time are those that set up the data structures used by forwarding time algorithms; route time computations happen relatively infrequently (on the order of every millisecond) compared to the rate at which forwarding happens (on the order of nanoseconds). For this reason we are not targeting the route time operations which form the bulk of router software. In contrast, computations that happen in forwarding time are the small amounts of code that need to be performed

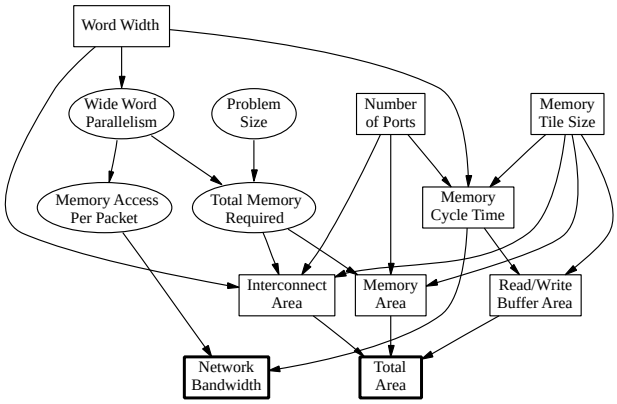

Figure 6: A graphical representation of the dependencies between hardware (drawn as boxes) and software (drawn as ellipses) when attempting to quantify wide-word pipelined memory network processor performance and area.

on every single packet that flows through the system. Making a chip that scales to these speeds while maintaining programmability and scalability is the real hardware challenge for backbone routers.

The tasks that need to be performed on each packet may be broken down into the following categories: Lookups, QoS, Classification, and Security. We define our benchmarks as a set of requirements, such as performing an IP lookup or classifying a packet stream rather than limiting ourselves to a particular implementation. Standard network processing benchmark suites either focus primarily on end user or edge functionality [8, 16, 27], or tie directly to a set of source code, neither of which is suited to our end goal of backbone router deployment. While the following algorithms are by no means an exhaustive set, they do provide a coarse generalization of the different types of algorithms used in modern and next-generation core routers. We use these to show the broad applicability of our architecture rather than as a hard proposal for a new benchmark suite. For each of these four algorithms we provide a description of the functionality, briefly explaining how it may take advantage of wide word accesses, and examine how well the algorithm maps to our architecture in terms of its estimated performance. We begin with a more in depth look at IP lookup, as it is one of the canonical examples of network algorithms.

5.2.1 IP Lookup

The most fundamental activity that a router must engage in is the forwarding of packets. In order to determine which link any given packet must forward to, a lookup must be performed. IP lookup takes a stream of destination IP addresses[3] as input, and is tasked with finding the output port that each packet must be forwarded too. This is complicated by the rules for packet forwarding, which require a *longest prefix* operation.

We spend more time describing IP lookup than the other algorithms because we use IP lookup as an example of the various techniques we have developed, and with the hope of giving the reader an intuition behind the architectural and algorithmic deci-

[3] Each destination IP address is either 32-bits (IPv4) or 128-bits (IPv6)

Original		Port
$1\star$	$\rightarrow$	P1
$0\star$	$\rightarrow$	P2
$01\star$	$\rightarrow$	P3
$011\star$	$\rightarrow$	P4

Expanded		Port
$10\star$	$\rightarrow$	P1
$11\star$	$\rightarrow$	P1
$00\star$	$\rightarrow$	P2
$01\star$	$\rightarrow$	P3
$0111\star$	$\rightarrow$	P4
$0110\star$	$\rightarrow$	P4

Figure 7: IP lookup table before and after controlled prefix expansion. The table on the left is smaller, but the table on the right can be more efficiently traversed because all of the patterns are aligned on 2-bit boundaries which allow 2-bits to be traversed at a time as opposed to 1 in the unexpanded table

sions we examine. For a survey of IP lookup algorithms we refer the reader to [21].

The left table in Figure 7 is an example IP lookup table. Any packet that enters the system is matched against the rule set, finds the longest match, and forwards the packet to the listed port. For example, a packet going to the address 1111 in our example will go out on port 1 (P1) because it matches $1\star$, and it is the only match. If a packet was going to address 0110, it would match three different patterns, $0\star$, $01\star$, $011\star$, but only the longest pattern, in this case $011\star$, is used and the packet would be forwarded to Port 4 (P4).

One solution to this problem is a Content Addressable Memory, or CAM. While CAMs can work well in some instances, they have trouble scaling both to the number of prefixes in modern routing tables (due to the increased size as compared to SRAM) and in available bandwidth. Instead, most of today's high end routers use a tree-like representation of the list of prefixes, and then walk this tree at high speed in hardware. A Patricia-Tree [17] is an example of this sort of tree that many people are familiar with.

Using Wide Words: While the Patricia-Tree can be effective in lower speed software-only routing implementations, the state of network algorithms has advanced significantly since the creation of Patricia-Trees. Instead, several routers use a Multi-Bit Trie approach as is detailed in [24, 9]. In this approach, the lookup tree is expanded so that multiple bits can be searched in parallel. An example of this can be seen on the right side of Figure 7. In the expanded table, you can see that all patterns are aligned on 2-bit boundaries which allows the table to be searched 2 bits at a time. At the same time you can see that the functionality of the table has not changed. The address 0110 now matches $0110\star$ (and in addition $01\star$), but the effect is the same (being forwarded to Port 4).

In general you can align the entries on any n-bit boundaries, enabling the lookup of n address bits at a time, and reducing the number of tree accesses (and memory access) by a factor of n. This optimization comes at the cost of expanded table sizes; in this example the table has gone from size 4 to 6, and generally the larger the number of bits that are looked up in parallel, the larger the table will get.

Because the memory expansion of multi-bit tries is large, the tables can grow large (on the order of 100,000 prefixes) and the amount of on-chip SRAM is limited. Thus, many routers today use compressed multibit tries such as the Lulea [9] scheme. Such schemes compress away the redundant portions of multi-bit trie nodes by adding bitmaps indicating repeated values in a multi-bit node. Bitmap compression can reduce memory needs considerably, at the cost of a increased memory accesses per node to read the bitmaps.

However, our wide word architecture provides us with a simple alternative first described in [18]. We can represent all the prefixes (and their corresponding pointers to other nodes) that lie within a multi-bit trie node without expansion. Assuming a node fits in a wide word, the entire node can be read in one memory access and the destination address can be compared in parallel with all the prefixes in the word. The longest matching prefix within this node is selected and the corresponding pointer is followed to fetch the next multi-bit trie node in the tree. Essentially, we are doing a parallel associative lookup of the node's contents to finesse the need for expansion or bitmaps.

Co-exploration: In Figure 8 we can see the overall tradeoffs that can be made by varying the parameters of both the architecture and the width of word used by the multi-bit trie algorithm. The x-axis is area in square millimeters, while the y-axis is the network bandwidth that can be supported by each algorithm. Every set of parameters that we examined is shown as a point on the graph. Pareto-optimal points are highlighted as circles. For this work a set of parameters is considered Pareto-optimal if there exists no other measured set of parameters that has both smaller area and better performance. For a reasonably sized system of 100 square millimeters, our technique can support throughputs approaching 100 Gbits/s assuming 40-byte packets. The key to achieving these bandwidths is the tradeoff between the extent to which IP lookup can exploit wide word access by searching multiple bits at a time, and the circuit level realities discussed in Section 4.

To search the space of IP addresses with wide words, you need to double the length of your word to search one more bit at a time, so there is a logarithmic relationship between the word size and the total number of memory accesses that the algorithm will need to make. In Figure 9, we examine this tradeoff for a particular slice of the parameters. For each algorithm, we plot the points for the highest throughput for each word size for a memory with 1 port that is limited to an area of 50 square millimeters. The results show that for IP Lookup, a width of 128 bytes is the best balance for all of the constraints.

5.2.2 Quality of Service

Supporting quality of service (QoS) primitives is a standard requirement for any high speed router today. Based on packet classification at the input port, packets are separated into queues at the output port of the router. For example, all web traffic may be directed to one queue. QoS is provided by a scheduling policy which determines which queue is serviced next to send a packet on the output link.

The simpler scheduling functions allow queues to be prioritized (e.g., for Voice over IP), rate limits to be applied to some queues using so-called token buckets, and simple, weighted round-robin schemes such as Deficit Round Robin (DRR). A DRR scheduler services queues in round-robin order, allowing each queue to send a quantum Q of bits in each round-robin cycle. Since only whole packets can be sent, and packet sizes may not fit into the quantum Q, the DRR scheduler keeps track of the quantum not used by a queue in one opportunity and gives it

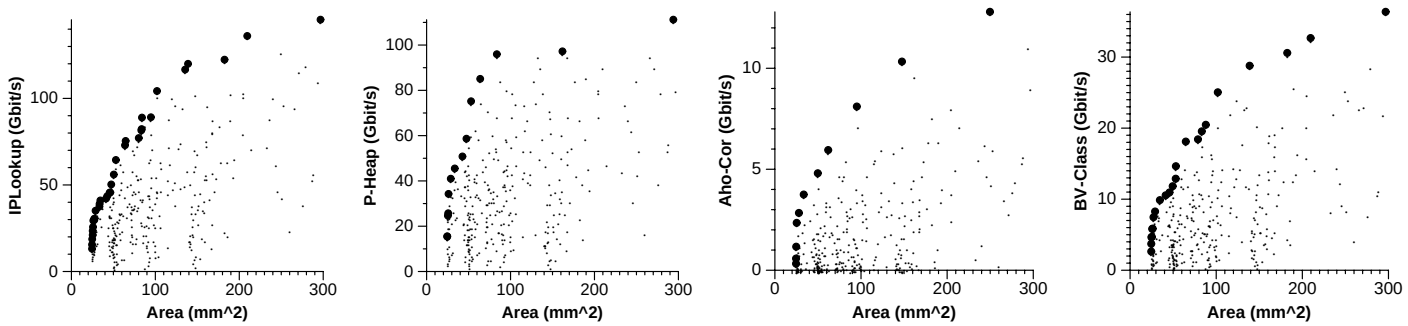

Figure 8: Tradeoffs for the different algorithms, as discovered by exploring the algorithm and architecture in unison for each algorithm individually. The x-axis is area in square millimeters and the y-axis is the network bandwidth that can be supported by each algorithm. All points examined are shown and Pareto-optimal points are shown as circles.

back in the next. All of these schemes maintain some per queue state on which the scheduler must base its decision as to which queue to send from next. The number of queues can be large; some routers have around 64,000 queues per output link.

More sophisticated schedulers, such as Weighted Fair Queueing [10, 3] and Virtual Clock [28, 29], have been designed to guarantee delays for delay-sensitive traffic, such as video, by implementing a scheme similar to earliest deadline schedulers. Thus each packet carries a deadline (or time stamp); the basic real time bottleneck is to select the lowest time stamp packet for transmission.

A natural data structure to compute the lowest time stamp is a heap. A heap is also useful in simple forms of QoS. For example, a heap can be used in rate limiting queues [11] and computing priorities, It can also be used in DRR to compute which queue to visit next in round robin order, while skipping over queues that have no data. An example of a data structure that allows the basic heap operations at high speeds and has a compact hardware implementation is the Pipelined-heap structure, or P-heap [4].

Using Wide Words: The P-heap is a structure that supports fully pipelined operation on a heap through the use of a side buffer that is used to communicate information between the stages, each of which is a level of the heap. The technique presented in [4] is designed with a hardware implementation in mind, but we adapt this technique to explore the algorithm space with our wide word memory design. We extend this work by supporting an even more aggressive heap structure that instead of being a binary heap (where each node has up to 2 children), can support heaps of arbitrary fanout. Through the use of wide words, we can support a higher out degree at each node in the heap, which means that the total height of the tree will be reduced. Note that wide word heaps, without the use of pipelining, have been implemented in custom ASICs in the past [11].

Co-exploration: The P-heap, unlike the Multi-bit Trie, is a balanced tree, which makes the analysis a bit simpler. In the case of this balanced heap, we can say that the height of the tree is strictly a function of the log of the number of nodes and the log of the fanout. However the structure is complicated by the fact that there are forwarding time writes. The major tradeoff for this structure is the fan-out of the nodes versus the memory and width constraints of the system. Here there is a constant size overhead associated with each node for storing the *active* and *value* fields (discussed in [4]), a counter which grows logarithmically in size with the number of nodes (the *capacity*), and a list of pointers which grows linearly with the fanout and logarithmically with the size of the memory. All of these must be examined in conjunction with the architectural tradeoffs.

Intuitively, by increasing the width of the memory we can make fatter nodes which then allows us to flatten the height of the heap. A flatter heap means less accesses to the heap are necessary in the worst case (where we need to access a path to the leaf nodes in the heap). Less memory access per packet means that more packets can be processed with a fixed amount of memory bandwidth, which means better throughput.

In Figure 8, we can see the results of performing the co-exploration of parameters. The pipelined heap application, as mapped to our architecture, can almost reach 100 Gbit/s with 100 square millimeters. This is similar to the IP lookup algorithm. The major difference between the way IP lookup and P-heap are implemented, is that in P-heap there is almost nothing to be gained in using more than 100 square millimeters. This is due to the fact that for P-heap there is expensive to increase the fan-out of the data structure (in terms of more overhead and control bits), which means that the algorithm benefits less from extremely wide words. Figure 9 shows this point for a slice of the parameters. As can be seen in the Figure, P-heap performs better with word sizes around 64 bytes. Above 64 bytes, the algorithmic gains of wider words do not balance out against the performance of the memory sub-system.

5.2.3 Security

Another problem that network infrastructure providers are encountering is the growing need to be able to track and block particular types of data on the network, such as worms. The ”Code Red II” worm, for example, is a piece of self-replicating malicious code that exploits a vulnerability in Microsoft IIS servers [6]. It has had an estimated world wide economic impact of over $2.6 billion dollars [1], and it can be stopped using a packet filter which checks for the string ”scripts/root.ex-e?” in the URI content along with a quick examination of the packet length.

There are many existing software systems that are designed to detect network intrusion or other malicious behavior. Unfortunately, the act of determining whether a particular packet is dangerous or not requires an examination of the payload of the packet. One operation in particular is essential, the ability to scan the packet contents for a set of strings which are known

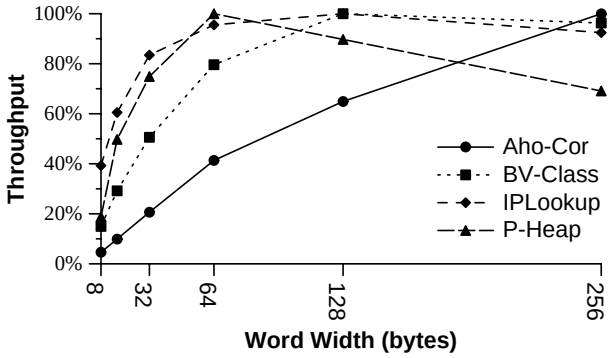

Figure 9: Graph of word width versus realized network bandwidth (as a percentage of the best achieved on a per-algorithm basis) for memory with 1 port, and limited to an area of 50 square millimeters.

to cause trouble. Readers may be familiar with efficient algorithms for string matching such as Boyer-Moore [5], which are designed to find a *single* string in a long input.

Our problem is slightly different, we are searching for one of a *set* of strings from the input stream. Instead of simply iterating a standard one-string matching algorithm, the set of strings that we are looking for can be folded into a single state-machine. This method, the Aho-Corasick algorithm [2], is what is used in the `fgrep` utility as well as the latest versions of the `Snort` [20] network intrusion detection system.

The Aho-Corasick algorithm works by building up a state machine that is fed the string to be searched. The state machine is generated by building up a tree of all the strings that need to be searched for, and a set of failure edges that are traversed when a search string on the path is not found.

This is another tree-like structure, but because of the failure edges (which point back to other parts of the tree), its traversal is not strictly tree-like. Another major difference between this algorithm and the others discussed thus far is the fact that this data structure must be accessed on *every single byte* of the packet, not just once for the packet header as in IP Lookup. This, as we will see, results in an order of magnitude less performance than the other three algorithms mentioned (as measured in total network bandwidth supported). However, we believe that this is a benchmark of growing importance and should be captured by our architecture.

Using Wide Words: To examine the behavior of the algorithm on real data, we generated the Aho-Corasick state machine for a set of strings used for actual intrusion detection and packet filtering. For this we used the default string set supplied with `Snort`, which includes, as part of it's rule base, a set of over 1000 suspicious strings resulting in an Aho-Corasick state machine with more than 10,000 nodes.

Because our data is no longer a proper tree (due to back edges), and because we want to ensure the worst case performance of the algorithm (across any packet stream), we now need to worry about the worst case cycle in the data structure. After building up a real state-machine from the `Snort` data, we found the major bottleneck to the system were several states at the top of the tree with very high fan-out. The root node in particular could have pointers to upwards of 150 possible next nodes, and the worst case cycle would always go through the root node. To combat this problem, we used our wide word resources to make these large nodes fit into less memory words. This handling of the large fan out nodes near the top of the state machine benefits the worst case performance as the total number memory access are reduced.

Co-exploration: In Figure 8 we can see that our architecture, at 100 square millimeters, can only handle 8 Gbit/s. This poor performance (relative to IP lookup in terms of total supported bandwidth) is due to a combination of effects. By far the largest contributer to the poor performance is the requirement that one node must be traversed for every byte in the packet.

When we only have to worry about packet headers, life is much easier because we only need to do one operation (such as lookup the IP address) for every 40-bytes of network traffic. However, when we are forced to examine the payload, we need to process every byte of network traffic, which is time consuming by its very nature. The second effect is that our current mapping of the Aho-Corasick algorithm is not making full use of our wide-word architecture. We should be able to map large sections of the state machine to wide words, and developing better implementations of the Aho-Corasick algorithm is a topic of our future research.

5.2.4 Classification

In recent years the trend has been for routers to rely less on traditional destination-based forwarding and more on the idea of service differentiation. The main idea is that services, such as allocations of bandwidth or reduced latency handling, are determined by the *type* of traffic that is being handled. Determining what type of traffic a particular packet qualifies as, and hence what rules apply to it, is driven by the header fields in the packet. For example, the destination address, source address and TCP/UDP port numbers could be used to classify priority voice over IP data.

Classification of a packet is done with a set of rules, each with an associated priority. Each rule specifies the type of packet that it applies to, typically in the form of a range or wild-card, and the rule with the highest priority that matches is the only one that is applied. For example, one rule might match all traffic going from address $001\star$ to $0100\star$ on port 80 (web). Thus classification can be considered to be rule based forwarding, with priorities to handle conflicts. Even within the core, fairly large (e.g., 2000 rule) classifiers are commonly used for security. For example, many of the rules appear to be denying traffic that is from specified subnetworks and is directed at particular servers or subnetworks within an ISP.

Among the best performing classification algorithms are the HiCuts algorithm [13] and the Bit-Vector Linear Search (BVLS) [15]. The HiCuts algorithm is a decision tree which makes range checks at each node in the tree. It can benefit from a wide word architecture by merging multiple decision nodes into a single node to reduce tree height. While both algorithms can benefit from wide words, we chose to expose the tradeoffs using the conceptually simpler BVLS scheme. For a survey of classification schemes we refer the reader to [12].

Using Wide Words: In the BVLS algorithm, a separate trie

is created for each of the fields (destination address, source address, port numbers, etc.) that can be used for rule matching. At the leaves of each of these tries is a long bit vector. Each element (bit) in the vector corresponds to a rule (from the classification rule set), and the vector is a list of rules that are eliminated from possibly matching because they do not match in this particular field. All of the the tries are walked based on the packet header information, leaving us with a set of vectors (one for each field). Each vector is the set of rules that have been eliminated, and if we take the logical OR of this set of vectors, we will be left with one vector that tells us which rules have not been eliminated yet, and thus, are matched.

There are two places where we can use wide word access to speed the data structures involved. The first way is to speed the walking of the separate tries used for each of the fields. The wide word access can be used in a way similar to that described in Section 5.2.1 and we do not discuss the tradeoffs again here.

The other advantage to be had from wide words is in operating on the long bit vectors themselves. Because there is one bit in the vector for each rule, and we need to examine over 2000 rules, and the bit vector OR operation requires many accesses to the memory. Note, our 2000 rule data set is similar to a rule set used by a major ISP. Using wide words here is a fairly simple matter of needing less separate words per vector when the vectors are longer that the word length. By using the wide word, we can operate on more of the bits at a time, and as such, we achieve a near linear increase in the bandwidth from an algorithmic standpoint.

Co-exploration: In Figure 8 we can see that using the BVLS algorithm can support approximately 20 Gbit/s at 100 square millimeters. While this number is short of the goal of 40 Gbits/s we set for ourselves, there are several areas that can be improved in terms of the implementation of the algorithm. Our current implementation of the algorithm is limited primarily by the initial trie lookups, which is why the shape of graph is similar to that for IP lookups. In addition, we can further improve these numbers by using wide words to target the initial trie lookups (currently not done), or by using an algorithm such a HiCuts [13].

5.3 Combined Analysis

Now that we have examined how each of the algorithms map to our full architecture, it makes sense to break down the effect of each of our contributions. We examine the best performance that can be obtained with and without the presented techniques for a constrained area in order to show this breakdown. Figure 10 was generated by taking a hard area limit, in this case 100 square millimeters, and by trying to find parameters of the algorithms and architecture such that total bandwidth supported is maximized, but the total area budget is not over-spent. Because of this, all these configurations have close to 100 square millimeters.

There are 5 bars on the graph. The first 4 are bars, where we view each algorithm independently (as was done in the previous sections), show results for tuning the architecture individually for each algorithm. The first bar, labeled *Standard*, is the base case where we assume a fairly simple algorithm that is limited to using 8-bytes of memory for each node, and where the memory access time is similar to a standard on-chip memory (with the decode logic separated from the actual read). While this base

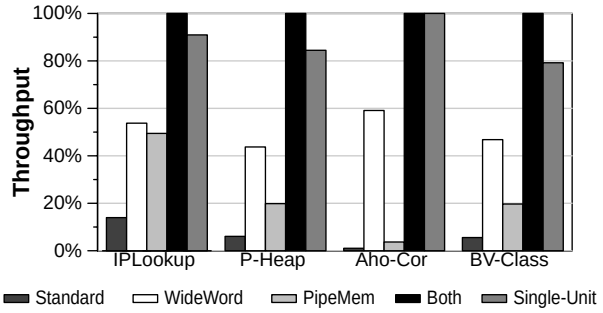

Figure 10: Examination of the best configurations found under different assumptions. The first 4 bars assume the architecture is tuned to a given architecture, while the last bar show what happens if a single architecture is tuned to the suite of applications. Results are normalized to the best configuration found for each *algorithm.*

case may or may not be an accurate representation of the state of the art, it does provide a point of comparison to show the importance of our other techniques.

The next bar on the graph shows the effect of having a memory system that has support for wide-word memory access, but it is not pipelined past the standard method. Note that, while this memory is not tiled into pipeline stages, internally the memory has been designed such that the internal delays are optimized using memory sub-banking to break up the otherwise exorbitantly long row and column lines. The third bar is the effect of just using a pipelined memory, without any use of wide words to deliver more parallelism or memory bandwidth.

The bar labeled *Both* is the most aggressive solution that we found for a given algorithm, but with the same constraint on the total area. Because this represents the best we could do using the techniques presented, and because the actual values for this can be seen in Figure 8, we have chosen to normalize all results for each algorithm to this bar. This bar represents, for a given algorithm, the best configuration tailor made to that algorithm.

In addition to these first four bars, which are tuned to each application separately, it is important to note the performance of the *best overall* performing configuration. The final bar, labeled *single-unit*, shows what happens if we pick a single best configuration, and map all of the algorithms to that configuration. This means that the architecture is tuned to optimize for the suite of applications rather than to a specific individual. The tuning is done by picking the architecture that has the highest geometric mean across all of the benchmarks. The difference between this bar and the *Both* bar shows the amount we lose from using a single system as opposed to systems custom tailored to each application, which is surprisingly little. This presents the hope that a *single* implementation could be used to achieve high performance from all of the applications we have examined.

6 Summary

This paper has focused on creating a programmable and scalable network processor solution for next generation backbone routers to achieve 40+ Gb/s. The two traditional network processor models we discussed have issues with scaling. The first

model uses independent processors with shared memory; the use of traditional memory designs optimized for latency limits the achievable memory throughput. The second model uses pipelined processors with partitioned memory to obtain high memory throughput. However, the second approach does not support writes and does not balance memory across partitions, both of which are important for router functions such as QoS and lookups.

To solve these issues we propose using a unified pipelined wide word memory. One of our main contributions is to achieve high memory throughput for a single, unified memory by internally pipelining the memory. This allows our design to process many transactions at the same time. In addition, the use of wide words allows our design to increase memory bandwidth, leveraging the fact that many network algorithms can be sped up using wide words.

Another contribution of this work is our co-exploration of the algorithm design with the design of the wide word pipelined memory architecture. We performed this co-exploration for several important network algorithms, which include IP lookup, QoS, Security, and Classification. We performed co-exploration by creating a parameterized version of these network algorithms and data structures that can exploit our wide word architecture. We then examined the tradeoffs between four different memory configurations varying the memory bandwidth achieved, word width and overall memory size. Through this co-exploration, we showed that a programmable architecture can efficiently exploit behavior inherent to many common network algorithms, and this is achieved by using our wide word pipelined memory. In particular, we found that a single set of parameters worked nearly optimally across all the IP functions we examined, suggesting that these parameters could be used as the basis for a 40 Gbps network processor design.

Acknowledgments

We would like to thank Jeremy Lau, Kristina Sherwood, and our anonymous reviewers for providing useful comments on this paper, and Nathan Tuck for his feedback on both the writing and analytical models. This work was funded in part by Semiconductor Research Corporation grant No. SRC-2001-HJ-897, NSF grant No. ANI-0074004, a grant from NIST on the Sensilla Project, and a grant from Intel.

References

[1] Malicious code attacks had $13.2 billion economic impact in 2001. Computer Economics, Jan 2002.

[2] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[3] Jon C. R. Bennett and Hui Zhang. Hierarchical packet fair queueing algorithms. *IEEE/ACM Transactions on Networking*, 5(5):675–689, 1997.

[4] R. Bhagwan and B. Lin. Fast and scalable priority queue architecture for high-speed network switches. In *Proc. of IEEE Infocomm*, March 2000.

[5] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):761–772, 1977.

[6] CERT/CC. Code Red worm exploiting buffer overflow in IIS indexing service DLL. CERT Advisory CA-2001-19, Jan 2002.

[7] Patrick Crowley and Jean-Loup Baer. A modeling framework for network processor systems. In *in Network Processor Workshop in conjunction with Eighth International Symposium on High Performance Computer Architecture (HPCA-8)*, Cambridge, MA, February 2002.

[8] Patrick Crowley, Marc E. Fiuczynski, Jean-Loup Baer, and Brian N. Bershad. Workloads for programmable network interfaces. In *IEEE 2nd Annual Workshop on Workload Characterization*, Austin, Texas, October 1999.

[9] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. Small forwarding tables for fast routing lookups. In *Proceedings of SIGCOMM*, pages 3–14, 1997.

[10] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queuing algorithm. In *Proc. SIGCOMM '89*, September 1989.

[11] Z. D. Dittia, G. M. Parulkar, and J. R. Cox. The APIC approach to high performance network interface design: Protected DMA and other techniques. In *Proc. of IEEE INFOCOM '97*, April 1997.

[12] P. Gupta and N. McKeown. Algorithms for packet classification. *IEEE Network Magazine*, 15(2), 2001.

[13] Pankaj Gupta and Nick McKeown. Packet classification using hierarchical intelligent cuttings. In *Proceedings of Hot Interconnects VII*, August 1999.

[14] Christoforos E. Kozyrakis, Stylianos Perissakis, David Patterson, Thomas Anderson, Krste Asanović, Neal Cardwell, Richard Fromm, Jason Golbus, Benjamin Gribstad, Kimberly Keeton, Randi Thomas, Noah Treuhaft, and Katherine Yelick. Scalable processors in the billion-transistor era: IRAM. *Computer*, 30(9):75–78, 1997.

[15] T. V. Lakshman and Dimitrios Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *Proceedings of SIGCOMM*, pages 203–214, 1998.

[16] Gokhan Memik, William H. Mangione-Smith, and Wendong Hu. Netbench: A benchmarking suite for network processors. In *IEEE International Conference Computer-Aided Design (ICCAD)*, 2001.

[17] D. R. Morrison. Patricia - practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, October 1968.

[18] Radia Perlman. *Interconnections: Bridges, Routers, Switches, and Internetworking Protocols*. Addison-Wesley, 2nd edition, October 1999.

[19] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo Lopez-Lagunas, Peter Mattson, and John D. Owens. A bandwidth-efficient architecture for media processing. In *31st International Symposium on Microarchitecture*, pages 3–13, November 1998.

[20] Martin Roesch. Snort – lightweight intrusion detection for networks. In *Proceedings of LISA'99: 13th Systems Administration Conference*, pages 229–238, November 1999.

[21] M. Sanchez, E. Biersack, and W. Dabbous. Survey and taxonomy of IP address lookup algorithms. *IEEE Network Magazine*, 15(2):8–23, 2001.

[22] Niraj Shah. Understanding network processors. Ver 1.0, September 2001.

[23] Tammo Spalink, Scott Karlin, Larry L. Peterson, and Yitzchak Gottlieb. Building a robust software-based router using network processors. In *Symposium on Operating Systems Principles*, pages 216–229, 2001.

[24] V. Srinivasan and G. Varghese. Fast address lookups using controlled prefix expansion. *ACM Transactions on Computer Systems*, 7(1):1–40, February 1999.

[25] Kevin Thompson, Miller Gregory J, and Rick Wilder. Wide-area internet traffic patterns and characteristics. *IEEE Network Magazine*, 11(6):10–23, November 1997.

[26] Tilman Wolf. A network processor performance and design model with benchmark parameterization. In *in Network Processor Workshop in conjunction with Eighth International Symposium on High Performance Computer Architecture (HPCA-8)*, Cambridge, MA, February 2002.

[27] Tilman Wolf and Mark A. Franklin. CommBench - a telecommunications benchmark for network processors. In *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software*, pages 154–163, April 2000.

[28] L. Zhang. Virtual clock: A new traffic control algorithm for packet switching networks. In *Proceedings of SIGCOMM '90*, 1990.

[29] L. Zhang. Virtual clock: A new traffic control algorithm for packet switching networks. *ACM Transactions on Computer Systems*, pages 101–125, May 1991.