Computer Sciences Department

Flexible Lookup Modules for Rapid Deployment of New Protocols in High-Speed Routers

Lorenzo De Carli
Yi Pan
Amit Kumar
Cristian Estan
Karthikeyan Sankaralingam

Technical Report #1658

June 2009

UNIVERSITY OF
WISCONSIN
MADISON

Flexible Lookup Modules for Rapid Deployment of New Protocols in High-speed Routers

Lorenzo De Carli Yi Pan Amit Kumar Cristian Estan Karthikeyan Sankaralingam
University of Wisconsin-Madison
{lorenzo,yipan,akumar,estan,karu}@cs.wisc.edu

ABSTRACT

New protocols for the data link and network layer are being proposed to address limitations of current protocols in terms of scalability, security, and manageability. High speed routers and switches that would need to implement these protocols traditionally perform packet processing using ASICs which offer high speed, low chip area, and low power. But with inflexible custom hardware, the deployment of new protocols could happen only through equipment upgrades. While newer routers use more flexible network processors for data plane processing, due to power and area constraints lookups in forwarding tables are done with custom lookup modules. Thus most of the proposed protocols can only be deployed with equipment upgrades.

To speed up the deployment of new protocols, we propose a flexible lookup module, PLUG (Pipelined Lookup Grid). We can achieve generality without loosing efficiency because various custom lookup modules have the same fundamental features we retain: area dominated by memories, simple processing, and strict access patterns defined by the data structure. We implemented IPv4, Ethernet, Ethane and SEATTLE in our dataflow-based programming model for the PLUG and mapped them to the PLUG hardware which consists of a grid of tiles. The throughput, area, power and latency we achieve are close to those of specialized lookup modules.

1. INTRODUCTION

The current Internet relies extensively on two protocols designed in the mid-'70s: Ethernet and IPv4. With time, due to needs not anticipated by the designs of these protocols, a number of new protocols, techniques and protocol extensions have been deployed in routers and switches changing how they process packets: Ethernet bridging, virtual LANs, tunnels, classless addressing, access control lists, network address translation, etc. Yet, the existing infrastructure has many acute shortcomings and new protocols are sorely needed. Data link and network layer protocols have been proposed recently to improve scalability [23, 29], security [51, 53, 15, 14], reduce equipment cost [2, 25], to ease management [23, 28, 14], or to offer users more control over their traffic [44, 52]. The two main factors that slow down the deployment of new protocols are the inevitable tussle of the various stakeholders [19] and the need for physical equipment upgrades.

The use of new equipment is a necessity for changes at the physical layer such as switching to new media or upgrades to higher link speeds. But data link layer and network layer changes do not necessarily require changes to the hardware and can be accomplished through software alone if the hardware is sufficiently flexible. Our goal is to enable such deployment of innovative new data plane protocols without having to change the hardware.

Sustained efforts by academia and industry have produced mature systems that take us closer to this goal. The NetFPGA project's [39] FPGA-based architecture allows one to deploy experimental changes to the data plane into operational backbones [11]. While FPGAs are an ideal platform for building high speed prototypes for new protocols, high power and low area efficiency make them less appealing for commercial routers. Many equipment manufacturers have taken the approach of developing network processors that are more efficient than FPGAs. For example Cisco's Silicon Packet Processor [22] and QuantumFlow [18] network processors can handle tens of gigabits of traffic per second with packet processing done by fully programmable 32-bit RISC cores. But for efficiency they implement forwarding table lookup with separate hardware modules customized to the protocol. Many of the proposed new protocols use different forwarding table structures and on these platforms, they cannot be deployed with software updates (without degrading throughput). Thus we are left with the original problem of hardware upgrades for deploying new protocols.

In this paper, we propose replacing custom lookup with flexible lookup modules that can accommodate new forwarding structures thus removing an important impediment to the speedy deployment of new protocols. The current lookup modules for different protocols have many fundamental similarities: they consist mostly of memories accessed according to strict access patterns defined by the data structure, they perform simple processing during lookup, and they function like deep pipelines with fixed latency and predictable throughput. We present a design for a general lookup module, PLUG (Pipelined Lookup Grid) which builds on these properties to achieve performance and efficiency close to those of existing custom lookup modules. Instead of using a completely flexible hardware substrate like an FPGA, PLUGs contain lightweight programmable microcores that perform the simple processing needed during lookup. Data structured are spatially layed out and PLUGs move the computation required by lookups to the relevant portions of the data structure. Our contributions in this paper are as follows:

- Programmable lookup modules that enable changing data plane protocols in high-speed routers and switches without hardware upgrades (Section 2);
- A dataflow-based programming model for representing forwarding tables (Section 3);
- An implementation of the forwarding tables for many existing and new protocols by using this model and a discussion of other possible uses (Section 4).

In Section 5, we outline a scalable tiled architecture that implements the programming model. Section 6 presents a static scheduling approach that avoids internal resource conflicts, simplifies hardware, and guarantees predictable throughput. Section 7 has a pre-

Protocol	L1 miss per lookup	Interconnection network bandwidth (Gbytes/sec.)	
		Software	Lookup module
Ethernet	8	256	6
IPv4	1.5	48	4
Seattle	6.5	208	6
Ethane	80	2560	27

Table 1: On-chip network requirements.

liminary evaluation of the efficiency of the proposed architecture.

2. A CASE FOR FLEXIBILITY IN LOOKUP MODULES

Since network processors are programmable, it is plausible to implement forwarding lookup for new protocols and even existing ones like Ethernet and IP directly on the network processor sharing a single in-memory copy of the forwarding table between all cores. While appealingly simple, such a solution has performance and cost disadvantages and high-speed routers do not adopt this approach. Such lookups would access multi-megabyte data structures with multiple reads from the data structure for a single lookup. Level-1 caches which are 8KB to 64KB in size are too small but Level-2 caches can capture the working set. However, the interconnection network required to connect the cores to a shared L2 cache holding the forwarding table would be extremely hard to build, require tremendous bisection bandwidth, and would be too slow. As the number of cores on chip increases it gets harder to scale this interconnect. We analyzed how such a system would perform by measuring the traffic needed between L1 and L2 caches for software lookups on an Intel Core2 processor with 32 byte L1 cache lines. These measurements were made by examining performance counters while running the protocols on representative data. Table 1 shows the average number of requests to the L2 cache from a single processor to satisfy a single lookup operation. To sustain a throughput of 1 billion lookups per second, we can then determine the number of requests the L2 cache must satisfy which in-turn is the bandwidth the interconnection network must provide. This ranges from 48 Gbytes/second to 2560 Gbytes/second as shown in the third column. In the fourth column, we show the bandwidth required to interface to a specialized lookup module which is at least an order of magnitude less.

Hence, high-speed network processors use separate lookup modules for forwarding tables. The lookup module receives the address from the network processor core and returns the lookup result a few cycles later. By using a lookup module the on-chip network connecting the cores to other resources has less traffic and thus a network with a smaller bisection bandwidth suffices. The local caches of the cores are not "polluted" with portions of the forwarding tables, so the cores can use smaller caches and less area. Also the overall latency of the lookup is reduced as a single roundtrip over the on-chip network is sufficient. Column 3 in Table 1 shows this approaches requires orders of magnitude lower bandwidth. The biggest drawback of this approach is the lack of flexibility. *A new lookup module must be implemented for each protocol.* We address this drawback by proposing a programmable lookup module.

2.1 Two Examples

To illustrate the state-of-art and show the potential for a flexible lookup module, we sketch how the forwarding tables for Ethernet and IP are implemented today (Figure 1). Based on the similarities and the differences we derive the mechanisms required for a universal lookup module.

Ethernet Forwarding: For Ethernet we need a lookup in a hash table using 48-bit Ethernet addresses as keys and port numbers as values associated with the keys. Figure 1a shows a simple hash table with 4 entries in each bucket. A key being looked up can be in any of the entries of the bucket it hashes to, so it must be compared against the keys stored in 4 entries. To reduce latency these comparisons are done in parallel by the custom lookup module from Figure 1b. The entries are divided among 4 memories, each holding one entry of each bucket. The memories are connected to local processing elements. During a lookup, the key and the number of the bucket it hashes to are broadcast to the four processing elements. Each one of them reads the entry corresponding to the bucket, compares the key stored there with the key being looked up and if the two match, it sends the port number onto the result bus. To increase throughput these operations can be pipelined.

IP Lookup: IP lookup requires the longest matching prefix operation which can be performed using a multibit trie as shown in Figure 1e. The trie is traversed from root to leaves and at each level two bits of the IP address are used to index into the current node[1]. At the location identified by these bits we may find the final result of the lookup (a port number) or a pointer to a node at the next level. Figure 1f shows how a custom lookup module can be organized: three memories, each with the nodes from a given level of the trie. Local processing elements read the correct node from the memory, perform the required processing and generate the data to be sent on the next link. The input link carries the IP address, the next one carries the remaining bits of the IP address together with either the final result if a match was found by the first processing element or the address of the node to be accessed at the next level. If the result is found earlier than the third level, later processing elements just pass it through without performing memory reads.

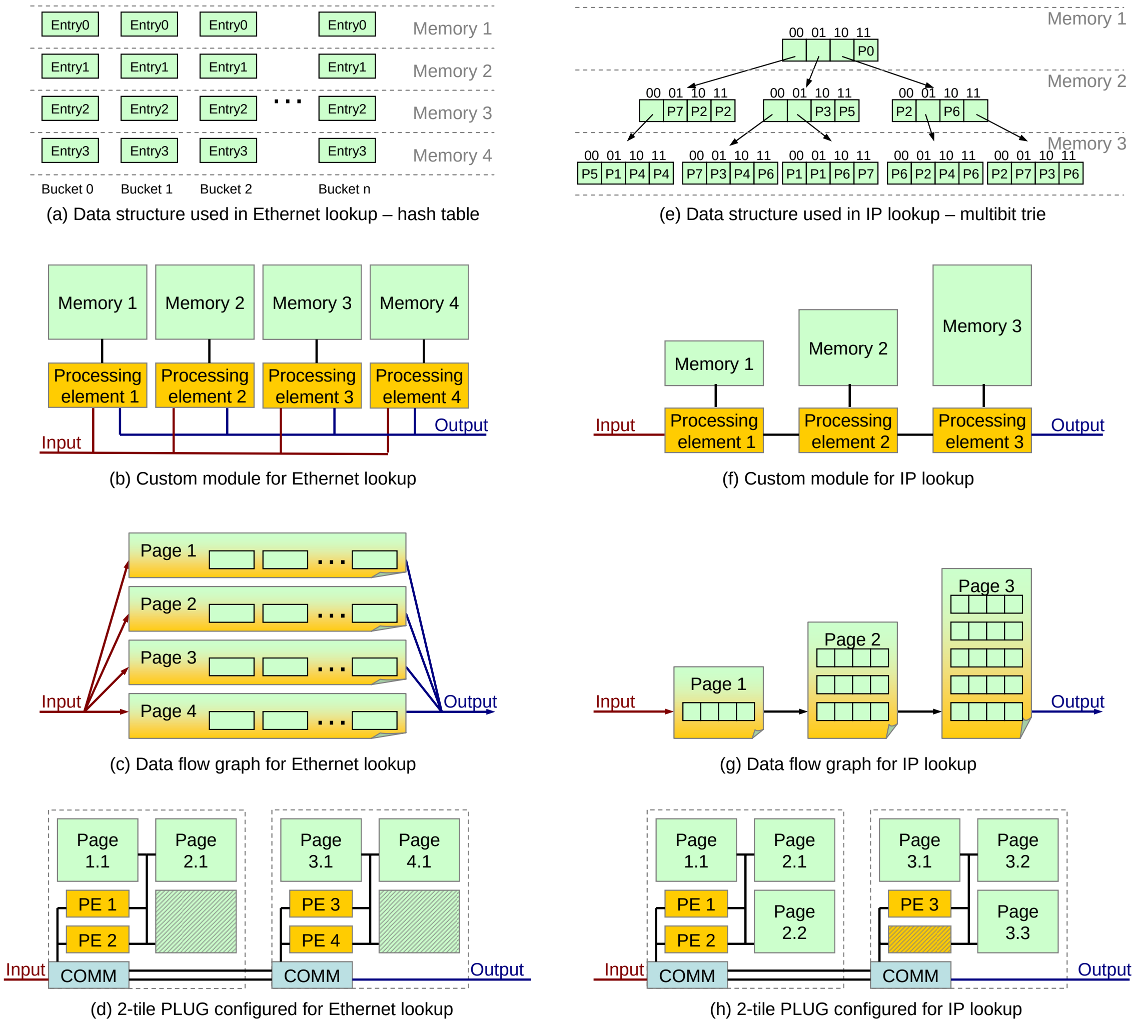
Similarities: The two custom lookup modules are fundamentally similar. Each have large memories connected to local processing elements. The processing elements perform simple operations, and each lookup follows an orderly succession of steps until it produces a result after a fixed number of cycles. But their overall structure differs because they implement different lookup algorithms. The custom lookup modules lack generality in three key respects: *the number and size of memories, the specific processing performed and the communication patterns supported*.

2.2 PLUG: A Universal Lookup Module

In developing the PLUG we first developed a programming model that enables the direct expression of the inherent structure in the lookup operation. *Lookup objects* describe this logical structure of the data in the forwarding table and the associated algorithms for lookups and updates. Conceptually the lookup objects are specified with data flow graphs (Figures 1c and 1g). The nodes of these data flow graphs are *logical pages* which represent portions of the data structure and the local processing steps for the data during lookups and updates. The directed edges of the data flow graphs denote the communication patterns. By extracting the structure, this programming model simplifies the architecture and programming.

The PLUG architecture implements the programming model using a modular design outlined in Figures 1d and 1h. More details of the architecture are in Section 5. It addresses the memory generality problem by having a large number of small memories that can be grouped together to implement memory regions of the desired sizes. Processing generality is achieved by using use lightweight 16-bit programmable processors. To accommodate any communication pattern, PLUGs use a multi-hop internal network that con-

[1] Since the trie in Figure 1e has three levels each corresponding to the two bits of the IP address, it holds prefixes of length up to 6.



(a) Data structure used in Ethernet lookup – hash table

(e) Data structure used in IP lookup – multibit trie

(b) Custom module for Ethernet lookup

(f) Custom module for IP lookup

(c) Data flow graph for Ethernet lookup

(g) Data flow graph for IP lookup

(d) 2-tile PLUG configured for Ethernet lookup

(h) 2-tile PLUG configured for IP lookup

Figure 1: Data structures used by the forwarding tables for Ethernet and IP, separate custom lookup modules for them, data flow graphs describing both algorithms and the mapping of both of them to the same 2-tile PLUG.

sists of communication modules at each tile and multiple direct links between all pairs of neighboring tiles. Figures 1d and 1h show how the two data flow graphs can be mapped to the same 2-tile PLUG. Multiple logical pages can be mapped to the same tile by allocating separate memories to each of them. If a logical page requires more memory than available on a tile, it can be mapped to multiple tiles. Note that through the small programs running on the lightweight processors, we don't just control the processing performed during lookups, but also the communication patterns and the number and location of memories implementing each logical page.

One of the distinguishing characteristics of the PLUG architecture is that it is globally statically scheduled to avoid resource conflicts. No two processing elements contend for the same memory or for the same communication ports. Three important benefits are: a) simplification of the hardware, b) same fixed latency guarantees for all lookup operations, c) processing a new lookup or update every cycle. The programming model rules explained in the next section allow such a statically scheduled architecture.

3. PROGRAMMING THE PLUG

A PLUG *lookup object* implements the main data structures of the forwarding table and the *methods* for accessing them. In this section we describe the primitives of the programming module: *data-blocks*, *code-blocks*, *messages*, and *logical pages*. We conclude with examples of hash table implementations using the model.

Data: *Data-blocks* are the primitive building blocks for logical pages and are small chunks of data that can be read with a single memory access. A logical page is a collection of data-blocks with similar roles in the data structure such that no method accesses more than one data-block in each page. This rule allows us to avoid conflicts for memory accesses.

Processing: The methods of the lookup object are broken into *code-blocks*. Each code-block is associated with one logical page and it can perform one memory access to read or write a data-block from that page. Each page has multiple code-blocks corresponding to the various methods of the lookup object. In practice, lookup methods are typically performance critical and most common, but additional methods are required to build and maintain the data structure.

Communication: *Messages* are used to communicate between code-blocks and the dataflow graph edges represent these messages. The execution of a code-block starts when the page receives a mes-

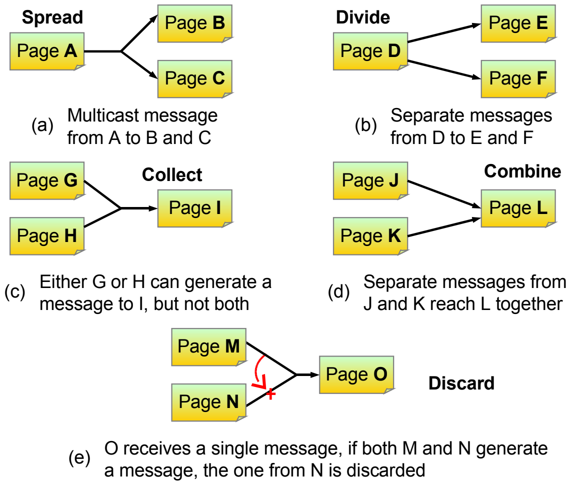

Spread
Page A
Page B
Page C
(a) Multicast message from A to B and C
Divide
Page D
Page E
Page F
(b) Separate messages from D to E and F
Page G
Page H
Collect
Page I
(c) Either G or H can generate a message to I, but not both
Page J
Page K
Combine
Page L
(d) Separate messages from J and K reach L together
Page M
Page N
Page O
Discard
(e) O receives a single message, if both M and N generate a message, the one from N is discarded

Figure 2: Communication patterns in data flow graphs.

sage which also indicates the type of code-block to execute. The entire context of the code-block consists of the data in the message and that in the data-block read from memory. Each data-block can send one or more messages to other pages and these must carry the entire context needed for the code-blocks they trigger. The execution of the lookup object is started by sending messages from the input interface of the PLUG. If the method produces results they emerge as messages at the output interface.

The communication patterns between the logical pages of the lookup object are described by the dataflow graph which is a directed acyclic graph. The nodes in this graph are logical pages with two special nodes of the input and the output interface. Each edge in the graph represents one message. In practice, a few patterns can be used to synthesize the complex dataflow graphs required for real protocols. Figure 2 shows these complex communication patterns, each of which requires some support in our software toolchain to implement. Simple features of the architecture are used to implement these communication patterns: the existence of multiple parallel network links (for the divide and combine patterns), the availability of multi-hop multicast messages (for the spread pattern), and static arbitration for deciding which message gets discarded on a conflict (for the discard pattern).

Implementing the model: The PLUG programming model allows a large variety of lookup operations to be implemented by a simple yet flexible hardware. The model imposes the following limitations: single memory access per page, strict adherence to the acyclic data flow graph, limits on the sizes of data-blocks, code-blocks, and messages.

Limitations: Some complex but infrequent operations (some updates, expiration of old entries, memory compaction, etc.) cannot be implemented as a single method of the lookup object. We separate such operations into an arbitrarily complex routine that runs on network processor cores and invokes one or more methods of the lookup object. These methods need to follow the same data-flow graph as the lookup methods. Thus an update to a page that is not directly connected to the input interface by an edge in the graph will have to go through one or more intermediate pages where simple code-blocks will just relay the update message. This is a generalization of the idea of “write bubbles” [8] used for updating IP lookup pipelines without disrupting lookups.

3.1 Hash Tables

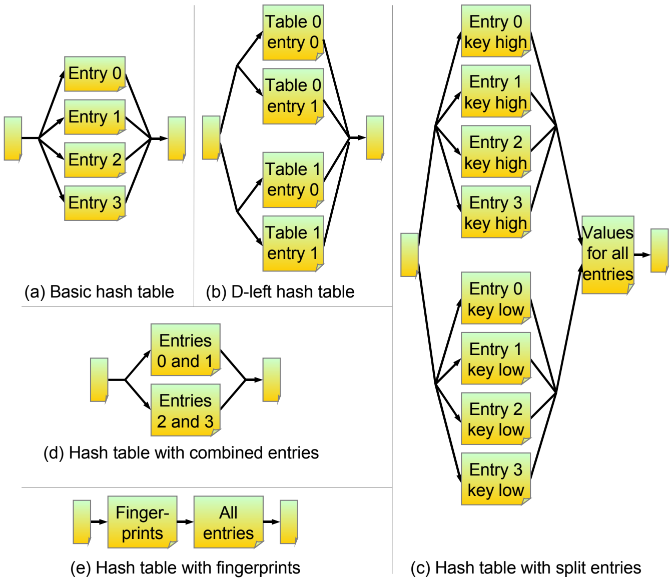


Figure 3: Data flow graphs corresponding to changes to the basic hash table design.

The lookup objects we implemented use many hash tables with wide ranges of values for parameters such as the number of entries, the size of the keys and the size of the values stored in entries. To simplify the presentation of the individual lookup objects we give below an overview of the 4 changes to the basic hash table we used. Figure 3 presents the data flow graphs corresponding to the basic hash table and the 4 modifications. Table 2 summarizes their advantages, disadvantages and applicability.

Multiple hash functions: In the basic hash table (Figure 3a), due to the randomness of the hash function, the buckets do not have the same number of elements in them. This presents a problem as when the number of used entries approaches the size of the hash table, some of the buckets will be full much before others. If a new key hashes to such a bucket it cannot be inserted, even though there are many empty entries in the hash table. To make such a situation unlikely, the hash table has to be run at a utilization of less than 100%. Using fewer, larger buckets allows better memory utilization, but it increases power consumption as we need more logical pages. D-left hashing [12, 13, 49] allows us to increase the memory utilization without increasing the number of pages. We use two tables with different hash functions and on insertion hash to both and insert in the table where the bucket the key hashes to is emptiest. This makes it less likely that any buckets will have significantly more entries occupied than the average. The example from Figure 3b implements d-left hashing. It uses two separate messages for the groups of pages implementing the two tables because different hash functions are used and the buckets in the two tables have different positions.

Splitting: In some cases (e.g. Ethane) the size of the key is larger than the maximum data-block size we can support. We can accommodate such hash tables by splitting the entries among multiple pages. In the example from Figure 3c we split the keys into a high key and low key stored on different pages. We send the two portions of the key to the two separate groups of pages and each page performs a check for its half of the key only. The last page keeps the values for all entries and it reads the value associated with the key only if both halves of the key successfully match indicated by

Modification	Applicability	Advantage	Main disadvantages
Larger buckets	always	better memory utilization	higher power
D-left hashing (b)	always	better memory utilization	multiple input messages
Split entries (c)	large entries	can fit keys larger than data-block	higher power, latency
Combined entries (d)	small entries	lower power	higher latency
Fingerprints (e)	with d-left hashing	lower power	higher latency

Table 2: The advantages and disadvantages of various modifications to the basic hash table.

it receiving two messages. There is one subtle possibility for false positives that we must check: the two halves of the key may have matched different entries of the bucket, and in this case the lookup must fail.

Merging: In other cases (e.g. Ethernet) the size of a hash table entry is smaller than half the size we can afford for a data-block. Combining two entries in a data-block as shown in Figure 3d allows us to reduce the number of pages (and hence power), but it increases latency slightly because the code blocks need to perform two key comparisons instead of one.

Fingerprints: A further opportunity for power saving is the use of fingerprints [12] together with checks for exact matches for the key as shown in Figure 3e. A first page contains small fingerprints (say 8 bits) which are computed together with the hash function and must be different for each of the keys in a bucket. All fingerprints for a bucket are read and compared against the fingerprint of the key being looked up. In the next page we can read directly the only entry for which the fingerprint matches (if any) to perform the check whether the keys match. This technique reduces power requirements significantly because we do not compare against more than one key in a bucket, but it increases latency because of the processing in the fingerprint page. The use of fingerprints introduces new conflicts as entries with the same fingerprint cannot be stored in the same bucket even if the keys differ. When fingerprints are used in combination with d-left hashing this is not that big a problem as the conflicting entry can be moved to the other table.

4. LOOKUP OBJECTS FOR PROTOCOLS

4.1 Ethernet

We implemented Ethernet forwarding with learning, but without VLAN support. The lookup object is a hash table with 4 entries in each bucket, d-left hashing (2 tables) and combined entries (2 per data-block). It uses a total of 4 logical pages. Each entry has 64 bits: a 48-bit Ethernet address, a valid bit, a 12-bit port number and a 3-bit timestamp counter. If the key passed to the lookup method matches a valid entry, the port number and timestamp are returned. For each unicast Ethernet packet two instances of the lookup methods are invoked for destination and the source. If the lookup on the source finds no match, we insert the address in the hash table and thus learn the port through which it is reachable. We keep outside the PLUG a secondary data structure summarizing which entries are used (valid) and which are not and use it to determine in which position to insert a new entry in.

We manage the expiration of the entries through the 3-bit timestamps. The system provides a coarse 3-bit “current time” variable whose value is stored in every newly created entry. When the lookup on the source address of a packet returns a timestamp that is older than the current time, we invoke an update method for updating only the timestamp in that entry. A background task periodically reads the valid entries one by one and invalidates those whose timestamp is too old. Our coarse 3-bit timestamps generalize the “activity bit” [14] used for garbage-collecting inactive entries and allow more accurate control of the actual amount of time an entry stays inactive before removal.

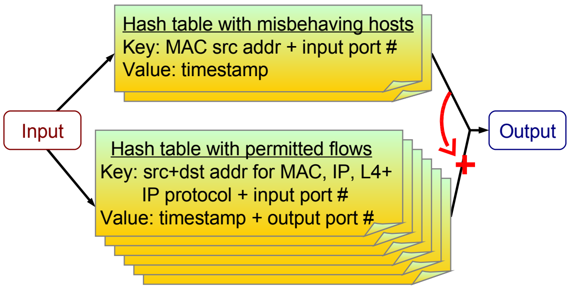

Hash table with misbehaving hosts
Key: MAC src addr + input port #
Value: timestamp
Input
Output
Hash table with permitted flows
Key: src+dst addr for MAC, IP, L4+
IP protocol + input port #
Value: timestamp + output port #

Figure 4: Data flow graph for Ethane lookup object.

4.2 Ethane

In Ethane [14] a Controller host explicitly builds the forwarding tables of all switches. The forwarding tables consist of two types of entries: flow entries which indicate how the switch should forward each allowed flow and per host entries indicating misbehaving hosts whose packets should be dropped. The lookup object (Figure 4) uses two separate hash tables for the two types of entries. If the lookups in both the hash tables succeed, only the result from the misbehaving host table reaches the output interface. We implemented the specific types of hash tables proposed in the original Ethane paper: both use d-left hashing with two tables and one entry per bucket. The key in the flow table is more than 26 bytes long, so we used the split entries (Figure 3c) since the specific instantiation of the architecture we consider limits the size of data blocks to 16 bytes. The lookup object has a total of 8 pages. By using larger buckets and fingerprints (Figure 3e) the memory utilization could be further improved without using more pages.

The Ethane flow table entries contain byte and packet counters which we store in arrays outside the PLUG. The reason is that the basic PLUG programming model mandates that each code block perform a single memory access, but incrementing counters requires two: a read and a write. The lookup method identifies the position of the entry and no further searching is required to locate the counters to increment. In Section 4.6 we show how the constraint of a single memory access per code block can be relaxed if lookups arrive at a rate lower than one per cycle. With a rate of one lookup every 6 cycles we could accommodate two counters per entry in the PLUG. For the specific PLUG we evaluate in this paper this translates to 167 million lookups per second which is more than the rate required for forwarding 64-byte packets at line rate for eight 10Gbps links.

To support multicast, in Ethane the flow table entries need to store the set of ports through which the packets of a given flow should be forwarded (note that the flow identifier also includes the

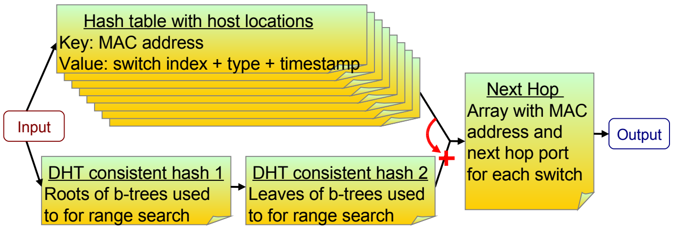

Hash table with host locations
Key: MAC address
Value: switch index + type + timestamp
Input
DHT consistent hash 1
Roots of b-trees used to for range search
DHT consistent hash 2
Leaves of b-trees used to for range search
Next Hop
Array with MAC address and next hop port for each switch
Output

Figure 5: Data flow graph for SEATTLE lookup object.

input port). Our implementation can be easily extended to support multicast for switches with a small number of ports (the Ethane paper reports implementing switches with 4 ports) by storing in the flow table a bitmap with the ports to which the packets should be sent. But since the size of the bitmap is the number of ports, for switches with more than a few dozen ports more radical changes to the lookup object would be required to avoid a big increase in memory usage.

4.3 Seattle

SEATTLE [29] switches participate in a routing protocol to compute shortest paths between each pair of switches. The data plane forwarding table of each switch has a table with the next hop to use for reaching any of the other switches. Endhost interfaces are reachable through a single switch that connects them to the SEATTLE network. Individual switches do not store forwarding entries for each remote host, they just keep a cache of popular destinations storing the switch they are reachable through (not the port they are reachable through). This way the cache needs not be invalidated when topology changes affect the path to switches that connect cached destinations. To locate unknown hosts, switches use a one-hop DHT in which each switch is responsible for storing indefinitely entries for all hosts hashing to a certain portion of the DHT ring. If a switch receives a packet whose destination is not in its cache, it needs to forward it to the resolver switch responsible for the portion of the hash space it maps to.

Our lookup object for SEATTLE (Figure 5) implements three components: a hash table mapping host addresses to their locations, a next hop array with the next hop port for each switch and a DHT ring for looking up the resolver switch implemented as an array of two-level B-trees. The structure of the lookup object is such that one lookup per packet is sufficient. In rare cases we also need an update of the timestamps in the location table. To minimize latency, the DHT ring and the location table are looked up in parallel. The DHT ring always produces a result, but if the lookup in the location table is successful it overrides it. Hence we always read the next hop array, either to find the next hop to the switch connecting the destination of the packet, or to its resolver.

The next hop table is a simple array with the addresses and the next hops for all the switches. We do not need a hash table because the location table and DHT ring store the position in the array at which the information associated with a switch is. We also need to store the address of the switch in this array because it is needed to build the outer header when the packet is encapsulated to be tunneled to the destination switch.

The location table uses MAC addresses as keys. It stores many types of entries: the cache of popular remote destinations, addresses for which the switch is the resolver in the DHT, locally connected addresses and the addresses for all switches. Each entry contains the position in the next hop array for the connecting switch. It also stores a timestamp and a 2-bit type field which identifies which of these four categories the entry belongs to. There are different policies for updating entries of different types. For example entries are removed from the cache when a timeout occurs, whereas entries for which the switch is the resolver are kept until the DHT changes.

The DHT ring needs to find the switch with the hashed identifier nearest, but no larger than that of the address being looked up. We divide the hash space into 4096 intervals of equal size and for each interval we keep a B-tree with the hashed identifiers of all the switches that map to it. We chose the number of intervals so that we can use B-trees of depth 2. The size of B-tree nodes is 128 bits: five 8-bit keys, five 16-bit values pointing to the entries for the switches associated with the keys and an 8-bit counter for the number of keys actually used. We use one of the values in the root node to point to the entry of the switch with the largest ID in earlier intervals. The second level of each B-tree has 5 nodes statically assigned to it, but the number of children actually used varies.

4.4 IP Version 4

For IP version 4 we implemented two lookup objects that perform the "longest matching prefix" operation. The first one is a straightforward adaptation of the "Lulea" algorithm [20] which uses compressed multibit tries and achieves very compact sizes for the forwarding table, but updates are relatively slow. The second one is an algorithm using similar techniques that results in slightly larger forwarding tables, but supports fast updates. For brevity we only describe here the second lookup object, but we note that the main difference from Lulea is that it does not perform leaf pushing.

The lookup object organizes the IPv4 forwarding table as a three-level multi-bit trie with the root node covering the first 16 bits of the IP address and the nodes at the other two levels covering 8 bits. Uncompressed trie nodes consist of two arrays of size 65536 for the root and 256 for the other nodes. The first array specifies the port associated with the longest matching prefix (from among those covered by the node) and the second holds pointers to children. For nodes without children, we omit the second array. In the result array there are often port numbers that repeat (covered by the same prefix from the forwarding table), and in the child array there are many entries with a NULL pointer. The compression algorithm saves space by removing repetitions from the first array and removing NULL pointers from the second. We use two compression techniques (Figure 6): bitmap compression for "dense" nodes and value-list compression for "sparse" nodes. A node is sparse if the compressed array is below a certain size (8 in our implementation, 4 in Figure 6). Bitmap compression breaks the arrays into chunks of 32 values (8 in Figure 6) and builds 32-bit bitmaps to find the right entry within these arrays during lookup. In the first bitmap the bit is set for positions which differ from the next one and in the second the bit is set for non-NULL children. The lookup code-block processing the summary counts the number of bits set before the position to which the IP address being looked up is mapped to. With value-list compression (not shown in Figure 6) we keep a list of values for the 8 bits covered by the node for which the port number differs from that for the previous value. For the child array we keep a list of values that correspond to children. The lookup code-block performs binary search in the value-list array to find the correct index in the two compressed arrays.

4.5 Other Protocols

IPv6: We are currently implementing a lookup object for IPv6. Since the last 64 bits of IPv6 addresses are the host part, prefixes with lengths from 65 to 127 are invalid and we use a hash table for the last 64 bits. To reduce the latency of the lookup we divide prefixes into two pipelines that are looked up in parallel: one for

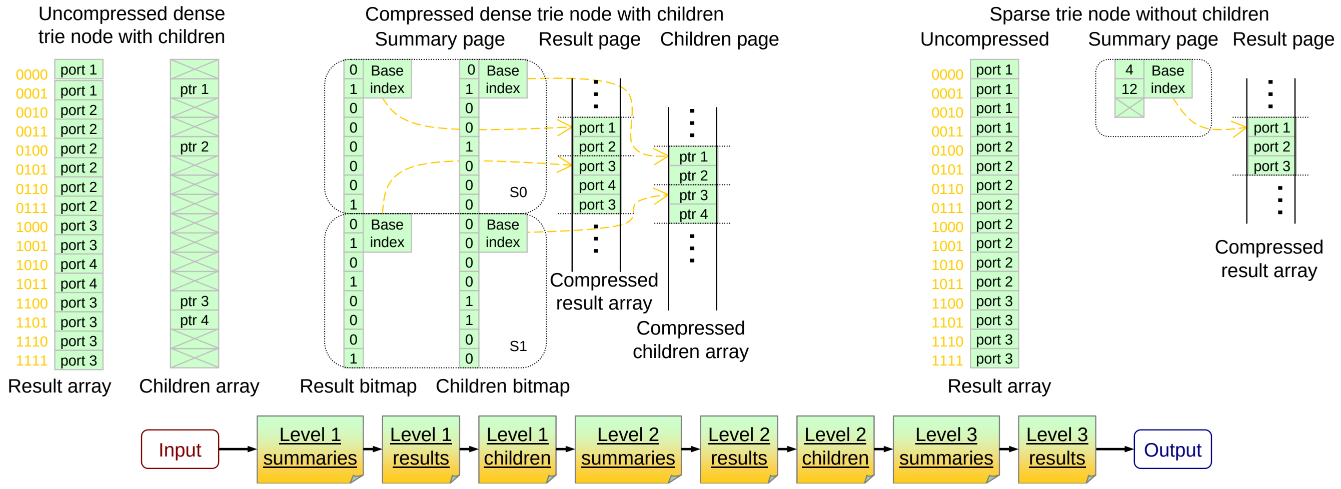


Figure 6: Compression technique and dataflow graph for the IPv4 lookup object.

prefix lengths of up to 48 and the other for prefix lengths of 49 to 64 and 128. The root of this second pipeline is a node with stride 48 implemented as a hash table. The latency of the IPv6 lookup is only 70% larger than than for IPv4, but it uses more than 3 times the number of pages (27) increasing power consumption.

AIP: The Accountable Internet Protocol [3] uses addresses composed of 160-bit cryptographic accountability domain (AD) addresses and 160-bit endpoint identifiers (EIDs). This helps solve many security problems that stem from the fundamental lack of accountability in the traditional IP. The core forwarding operation on AIP addresses requires looking up large hash tables with AD addresses or EIDs as keys and PLUG can implement this efficiently. Since 144 of the 160 bits are generated by cryptographic hash functions we can exploit their randomness to reduce the size of the actual keys stored in the hash table buckets. We can use some of these bits as the bucket identifier and others as fingerprint, thus the actual entry only needs to store the remaining bits.

PLayer: PLayer [28] is a data link protocol for data centers achieving more robust control of the traffic by using policy-aware switches to direct traffic through the right sequence of middle boxes. Forwarding is based on rules on the source of the packet and the flow 5-tuple (source and destination IP and port number and protocol). This is similar to the packet classification operation discussed in Section 4.6 and can be implemented on PLUG.

NAT: PLUGs are well suited for implementing network address translation with hash tables and arrays. We also considered a NAT-based architecture, IPNL [23]. Most of the required operations can be implemented by PLUGs. The one operation that is hard to implement directly on PLUG is lookups when keys are variable-sized such as for a cache of mappings from fully qualified domain names to IP addresses. A hash table with collision-resistant fingerprints of the FQDNs as keys would be straightforward.

4.6 Discussion

PLUGs can be used for other functions and they can be used in scaled up or scaled down modes which we discuss below.

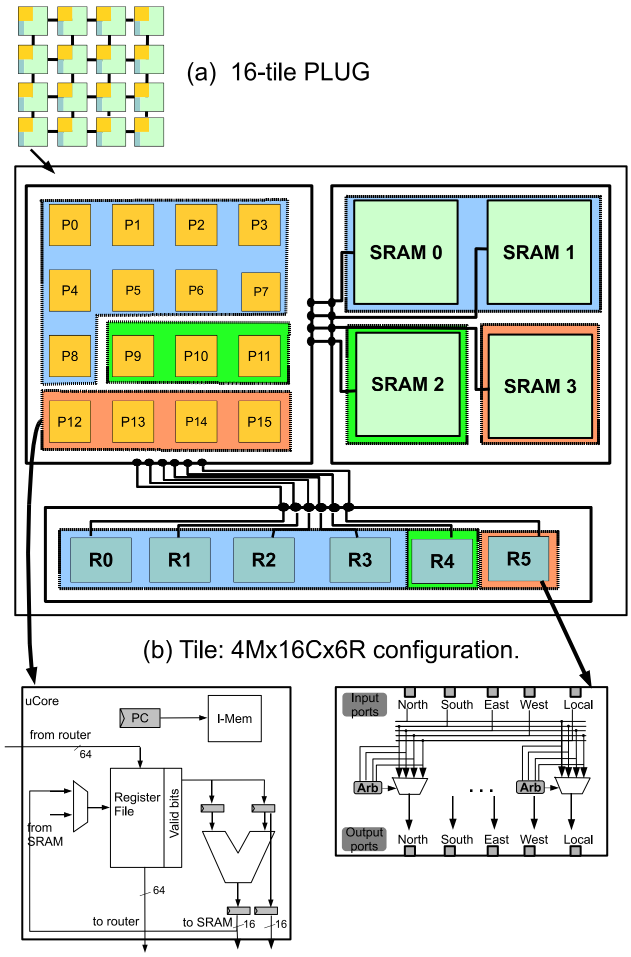
Packet classification: Another throughput-critical data plane operation involves matching packet headers against rules with ranges or prefixes for each of the header fields (source, destination and protocol for IP and source and destination port). TCAM-based solutions for the packet classification problem cannot be implemented on our low-power SRAM-based architecture, but many of the algorithmic solutions could work well. Algorithms that rely on processing the fields independently and combining the results through bitmap operations [35, 7] can make use of the parallelism of the PLUG programming model, the wide memory reads, and the ability to select the highest priority answers by discarding lower priority results. Other approaches relying on decision trees [26, 43] or forests of tries [5] can be naturally adapted to a pipeline and the flexibility of the programmable $\mu cores$ makes it easy to support the required operations on the tree nodes. While we have not implemented any of these algorithms on PLUG, we believe that packet classification can be one of the important uses for it.

Signature matching: Deep packet inspection often relies on DFA-based signature matching and compression techniques [33, 34, 9, 31] are often applied to reduce the memory used by transition tables. Lookup in these compressed transition tables is well suited for PLUG and we have an implementation using some of the proposed techniques. The latency of the PLUG can be a problem, but techniques that consume multiple bytes at a time [10] may alleviate it.

Per flow state: Features such as accounting, application identification and intrusion prevention are supported by modern network equipment and require per flow state. Hash tables with per flow state can be placed on the PLUG. With current technology PLUGs that can accommodate on the order of one million flows are feasible.

Scaling down: In settings where the target throughput is less than one lookup per cycle, the programming model can be extended in powerful ways. If the pipeline is only required to accept a new method invocation every X cycles, we can relax the limitation that each code block accesses the memory once. In fact each code block has a window of X cycles in which it can perform multiple memory accesses, for example to read increment and then write counters. The semantics of concurrent execution of multiple methods in the pipeline is still that of atomic sequential execution.

Scaling up: If the desired throughput is larger than one lookup per cycle, multiple independent pipelines with separate copies of the forwarding table can be used. The cost of the increase in throughput is an increase in memory requirements, but it can be reduced by taking advantage of the existence of popular destinations in the forwarding tables. We can split the forwarding tables into a small group of trie nodes or hash table entries that store popular destinations and a larger group of unpopular ones. The pipelines can have



(a) 16-tile PLUG

(b) Tile: 4Mx16Cx6R configuration.

(c) uCore and Router

Figure 7: PLUG Architecture Overview. Three virtual tiles to map three logical pages. (b) shows details for one tile.

separate copies of the popular entries but share one copy of the unpopular ones. Collisions could still occur in the shared pipeline, but they should be rare and one can recover by submitting again the lookup that did not complete due to the collision. Hence we can achieve performance benefits akin to those of caching in this statically scheduled architecture.

Limitations: PLUGs are not suitable for all data plane processing. For some tasks a traditional architecture (multiple processor cores with multiple levels of caches backed by DRAM) is better suited. There are three fundamental characteristics that distinguish tasks suitable for a PLUG: the ability to describe the task with an acyclic data flow graph, use of a data structure that can fit into SRAM, and poor locality in this data structure. Signature matching using DFAs involves cyclic data dependencies and hence the task cannot be described as an acyclic data flow graph; PLUG can be used as a lookup module invoked separately for each byte of input, but it cannot implement the whole computation. The second condition may not hold for equipment where per flow state is too large and a PLUG is able to accommodate only a fraction of the flow records. Examples of where the third condition does not hold are tasks such as parsing protocol headers where no data structure is used or protocols for which the data structure is small enough to fit in L1 cache [2].

5. PLUG ARCHITECTURE

Figure 7 shows the high level overview of the PLUG architecture. It is a tiled multi-core multi-threaded architecture with a very simple on-chip network connecting the tiles. Tiles provide three types of resources: computation shown by the grid of computation cores (called $\mu core$s), storage shown by the grid of SRAMs, and routers shown by the array of routers. The routers form an on-chip interconnection network connecting multiple tiles together to form the full PLUG chip. External interfaces to provide input and read output from the PLUG are extensions of this on-chip network. The architecture can support multiple external interfaces and we expect PLUGs to be implemented as standalone chips or integrated with other chips.

A key simplification of the architecture is that it is globally statically scheduled which is achieved by adhering to a set of rules in generating the code-blocks. In the remainder of this section, we describe the organization of each tile, the ISA, the $\mu core$ in each tile, and the on-chip network.

Tile: The tile's resources are virtualized as a memory array with M ports and thus allowing up to M accesses at a time, a router array with R ports thus allowing R input/output messages at a time, and a computation array with C cores. The PLUG architecture can be scaled along any of these dimensions, and the figure shows a 4Mx16Cx6R configuration. This virtualization allows the several logical pages and their associated code-blocks to be mapped to a tile. Thus one physical tile can be viewed as multiple logical or virtual tiles, where each such virtual tile has a subset of the resources. An example assignment is indicated by the coloring shown in the figure, where three logical pages are mapped to a single tile by constructing three virtual tiles colored blue, green, and orange. A set of programmer-visible special tile-registers are available to associate sets of cores with each memory port and router.

When a message arrives through a network, it triggers the execution of the code-block it refers to. The next available free $\mu core$ starts executing the code-block. If another message arrives in the next cycle, another $\mu core$ is used. When a code-block finishes executing, that $\mu core$ is free. The resource constraints dictate that no more than M logical pages can be assigned to a single tile and thus the maximum number of virtual tiles is M. Depending on the length of the code-blocks, different number of $\mu core$s are assigned to each logical page. The number of networks assigned to each virtual tile depends on the number of messages generated.

μCores: Each $\mu core$ is a simple 16-bit single-issue, in-order processor with only integer computation support, simple control-flow support (no branch prediction), and simple instruction memory (256 entries per $\mu core$). The register file is quite small, only sixteen 16-bit entries and it requires four consecutive entries to be read for feeding the router.

Router: Each tile also includes simple routers that implement a lightweight on-chip network (OCN). Compared to conventional OCNs, this network requires no buffering or flow control as the OCN traffic is guaranteed to be conflict-free. The router's high level schematic is shown in Figure 7c. The routers implement simple ordered routing and the arbiters apply a fixed priority scheme. They examine which of the input ports have valid data, and need to be routed to that output port. On arrival of messages at the local input port, the data gets written directly into the register file. Each network is 64-bits wide, and the software mapper assigns networks to the virtual tiles. The network message is a total of 80 bits, 16 bits of header information and 64 bits of data. The header information contains five fields: destination encoded as a X coordinate and Y coordinate, a 4-bit type field to encode 16 possible types of messages, a multicast bit which indicates the message must be delivered to some intermediate hops en route to the final destination and a 3-bit selector field.

This field is used to select between virtual tiles and to control which of the tiles on the path of a multicast message actually process it.

ISA: The PLUG ISA closely resembles a conventional RISC ISA, but with two key specializations. It includes additional formats to specify bit manipulation operations and simple on-chip network communication capability. The ISA uses 16-bit words and supports variable length loads that can read up to 128 bits and write them into consecutive registers. We decided on 16-bit words based on analysis of typical workloads.

5.1 Implementation Specification

To derive a PLUG configuration, we examined different protocols and current and projected future data sets. Our target was 1 billion lookups per second (depending on minimum packet size and on the number of lookups/packet, maximum traffic volume is between 160 Gbps and 600 Gbps) and less than 5 watts worst case power. We picked a technology design point in the future and chose a 32nm design process (which will be in volume production in 2010 [1]). Many of our data-sets required 2MB and to account for future expansion, we provisioned for 4MB on-chip storage. Based on area models we developed (explained below) a $21mm^2$ chip provides 16 tiles and a total storage of 4MB. Our workload analysis showed 32 cores, four 64KB banks, and 8 networks per tile meets our design goals. This 16-tile 4Mx32Cx8R configuration is the specification evaluated in this paper.

5.2 Modeling and Physical Design

We constructed area and power models for the PLUG architecture using widely accepted design tools and methodology. Our models include three components: SRAM, the $\mu core$s, and interconnection network.

Area: The SRAMs were modeled using CACTI 5.0 [48] - a standard SRAM modeling tool. We used single-ported SRAMs and to save power used LSTP memory cells which have low static power[2]. For modeling the $\mu core$ array, we used published processor datasheets and used the Tensilica Xtensa LX2 [47] as a baseline for our $\mu core$. This processor is a simple 32-bit, 5-stage in-order processor and occupies 0.206mm^2 built at 90nm. Projecting for 32nm technology and simplifying to a 16-bit data path, we scale down its area and our models project 0.013mm^2 . We conservatively assumed the interconnect's area is 10% of processor area. Based on this model, a single PLUG tile's area is 1.29 mm^2 of which 74% is SRAM.

Power: CACTI provides power measurements for SRAMs and we estimate worst case power by assuming the SRAM is accessed every cycle. We used processor data-sheet information about the Xtensa LX2 processor to derive the power consumption of our 32 $\mu core$ array. We model interconnect power per tile as 15% of processor ($\mu core$ array in our case) dynamic power for an active link, adapting the findings in [50]. The worst case power for a tile is 990 milliwatts.

Dynamic chip power for the PLUG is derived by considering the (maximum) number of tiles that will be activated during the execution of a method (*activity number (A)*) and considering and *average links active (L)*. Thus, worst case power can be modeled as A tiles executing instructions in all $\mu core$s and one memory access every cycle. We compute L based on the mappings of the lookup objects to the grid discussed in Section 6. The final chip power = [((memory leakage power per tile) + $\mu core$ leakage power per tile)) * (total number of tiles)] + [((dynamic memory power per tile) + $\mu core$ power per tile)) * (activity number)] + [(interconnect power per active link) * (average links active)]. This model is seeded with the *activity number (A)* for different protocols.

6. SCHEDULING AND MAPPING

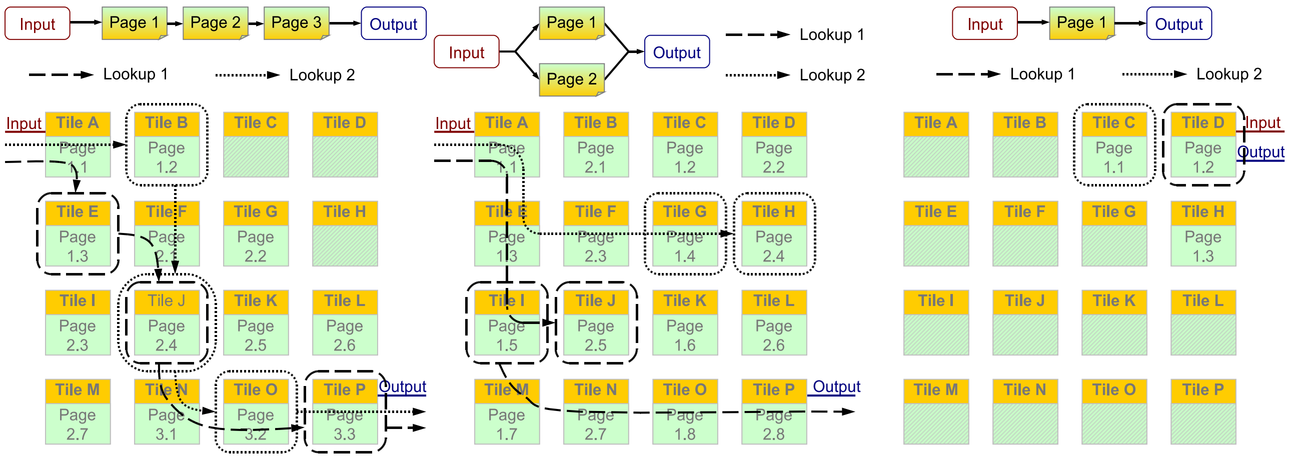
Earlier sections describe the lookup objects, we show here how the logical pages and code-blocks can be mapped to the actual physical resources. While we currently perform these operations manually, we envision that the compiler could eventually make all mapping decisions. A first step is to convert logical pages into smaller physical pages that can fit on an SRAM bank. Different logical pages can get mapped to a single physical tile and each logical page has its own set of SRAM banks and $\mu core$s. Second, the mapper also checks for resource conflicts - the number of instructions in the code-blocks that have been mapped to a tile must not exceed the total number of $\mu core$s available. Third, the mapper also enforces fixed delays as explained in detail below.

Code-blocks: Each tile can be executing multiple instances of code-blocks for the same logical page. While memories and networks are not shared between different pages mapped to the same tile, the cores that run code-blocks for the same page could get into conflicts when accessing memory or sending messages. We ensure that each code-block for a page performs the memory operation the same number of cycles after its start. Since the code-block instances running concurrently are started in different cycles, this ensures that no memory conflicts arise. Similarly the sending of messages occurs a fixed number of cycles after the code-block is started.

Different paths: Two lookup instances can take different paths through the grid. The time it takes for the result to emerge is the sum of the processing delays (which are the same for all lookups) and the total propagation delay. If the paths have different lengths the total latencies differ and conflicts can occur at the output interface (lookups initiated in different cycles emerge at the same time) or inside the grid. Figure 8a shows how adopting a propagation discipline can ensure that all paths have equal length. Since messages propagate only to the right or down, the propagation latency to every tile is the Manhattan distance from the tile connected to the input interface. The two lookups reach tile J through different paths, but their propagation delays are the same. Note that a conflict on the link between tile O and tile P is still possible: the propagation delays are the same, but the processing delays differ as lookup 1 has been processed by the first two pages, while lookup 3 has been processed by all three. To avoid these types of conflicts we use two separate networks. More generally this type of conflict arises because the two messages correspond to two different edges in the data flow graph: for lookup 1 the edge between pages 2 and 3 and for lookup 2 the edge between page 3 and the output. The propagation disciplines can ensure that messages corresponding to the same edge do not collide, but if messages corresponding to different edges can occur in the same area of the grid the edges must be mapped to different networks.

Multicast: Hash tables use "spread" edges (Figure 2a) to distribute the lookup to multiple logical pages. We implement these using multicast messages. Each multicast message takes a single path through the grid, but it may trigger code blocks at more than one tile. In Figure 8b we show the mapping of a 2-page hash table to a 16-tile grid. Each logical page is mapped to 8 tiles and the mapping ensures that each lookup will need to activate entries from the two pages mapped to neighboring tiles. To avoid triggering code blocks on intermediate tiles (e.g. tiles A and E for lookup 1) we use the selector field from the the message header. Code-

[2] The LSTP transistors trade off high on-currents for maintenance of an almost constant low leakage of $10pA/\mu m$ across technology nodes by using longer gate length, thicker gate oxide, higher threshold voltage, and higher V_{dd} [1].



(a) Avoiding conflicts with equal propagation delays

(b) Using multicast to implement "spread" edges

(c) Unequal propagation delays

Figure 8: Mapping lookup objects to the PLUG grid.

Protocol	Memory size (MB)	# of logical pages	# of code-blocks	Lines of code		# of tiles
				PLUG	Reference	
Ethernet	2	4	4	243	51	8
IPv4	1.4	8	26	450	330	8
Seattle	2	11	18	390	347	11
Ethane	2	8	15	1120	200	8

Table 3: Characterization of the lookup objects implemented on a 16-tile PLUG.

blocks are triggered only if the value of this field matches a local configuration register (similar to how multicast is implemented by Ethernet endhosts).

Unequal propagation delays: In some situations we cannot avoid unequal propagation delays. In Figure 8c we show the mapping of a simple one-page lookup object onto the tiles not used by the object from Figure 8a. Latency is minimized by using tile D for both the input and output interface, but this means that the propagation delay of lookup 2 is two cycles whereas for lookup 1 it is zero cycles. To avoid a conflict at the output interface we add two cycles of "padding" to the code blocks running on tile D.

7. EVALUATION

In this section, we evaluate PLUGs and their suitability and efficiency in supporting our suite of four protocols. We implemented each of the protocols using our software development stack. First, we present a quantitative characterization of the different protocols to demonstrate feasibility. We then describe the mapping of these protocols to the PLUG architecture and evaluate performance in terms of latency, area, and energy efficiency. For all experiments we use a 21 mm^2 4Mx32Cx8R 16-tile PLUG chip with 64KB memory banks.

7.1 Software Toolchain and Methodology

We implement PLUG lookup objects as C++ objects. For each lookup object, we first developed a reference implementation which directly implements the application without transforming it to the PLUG model but implements the same routines (lookup, updates, expiration of old entries, etc.). We use a C++ framework, which provides the following: 1) a logical-page data structure as an array data-type, 2) explicitly defined code-block functions to access these data structures, and 3) network messages that are routed between physical pages. Applications are implemented using this framework and executed by invoking methods and passing input network messages. Hash functions are computed by the network processor. Our framework executes the corresponding code-blocks on different pages and finally provides the result message which is verified by comparing to the reference implementation. For performance analysis, we hand-assembled code-block programs, based on the C++ implementation.

Protocol	Logical page name	bytes per data-block	page size (KB)	mem. banks per page	Latency (cycles)	
					Code block	Total
Ethernet	Buckets	16	512*4	8	10	10
IPv4	L1 summaries	12	24	1	10	82
	L1 results	2	34	1	6	
	L1 children	2	27	1	9	
	L2 summaries	8-14	609	10	20	
	L2 results	2	685	11	6	
	L2 children	2	7	1	9	
	L3 summaries	6-9	36	1	16	
	L3 results	2	26	1	6	
Seattle	HostLoc	9	144*8	3	14	49
	DHT Ring 0	16	64	1	20	
	DHT Ring 1	16	320	5	20	
	NextHop	8	512	8	9	
Ethane	Misbehaving	16	256*2	4	21	31
	Flow table 1	16	256*4	4	10	
	Flow table 2	16	256*2	4	10	

Table 4: Logical page characterization of the protocols.

7.2 Quantitative Results

We evaluate the cost of the generality provided by PLUG by comparing against idealized implementations for lookup modules for all the protocols we implemented. We also compare against NetLogic's NLA9000, a widely used custom lookup module for IPv4 that implements a proprietary algorithmic pipelined lookup.

Data sets: Table 3 describes the characteristics of the four lookup objects we implemented. For each protocol we used data-sets and

Protocol	Memory (MB)			Area (mm^2)		
	PLUG	Ideal	Overhead	PLUG	Ideal	Overhead
Ethernet	2.00	2.0	0%	9.8	6.6	49%
IPv4	2.00	1.4	43%	7.4	4.6	61%
Seattle	2.75	2.0	38%	13.5	7.2	88%
Ethane	2.00	2.0	0%	9.8	6.6	49%

Table 5: Memory and area characteristics.

traces that reflect deployed scenarios and/or future needs. Ethernet forwarding uses random 100K addresses, IPv4 uses an actual routing table with 280K prefixes, Seattle has been dimensioned to support 60K switches and for Ethane we used the guidelines from [14]. For all protocols, we see that the number of logical pages is quite small and that the number of lines of code to develop the PLUG implementations is similar to the reference implementation. The last column in the table also shows the number of tiles in the PLUG used.

Table 4 describes in detail the characteristics of individual logical pages. It shows the size of the data-blocks in each logical page and how many physical memory banks are used to schedule this page to the PLUG hardware.

Latency: Column 6 in Table 4 shows the latency in cycles of the largest code-block associated with each logical page of an application in the last column. These numbers vary between 6 and 20 cycles. The total latency is simply the sum of the code-block latencies along the dataflow graph's critical path and it varies between 10 and 82. These numbers do not include the propagation latency which is 8 cycles on our 16-tile grid.

Memory and area: Table 5 characterizes the memory occupied by the different implementations. During the mapping process logical pages are broken down into physical pages that will fit on a memory bank. Then, we decide which page is mapped to which tile and which bank. To avoid conflicts, certain banks may not get completely full because they contain a physical page smaller than the memory available in the bank. Occasionally we leave banks unused on tiles where we cannot map a new page because too many $\mu core$s are used by the pages already mapped to the tile. Column 2 in Table 5, shows the total memory (sum of SRAM banks) used in the different protocols accounting for these fragmentation and scheduling losses and compares to the actual memory required by the application. For Ethernet forwarding and Ethane which both consist of very regular hash tables, there is no overhead. In the other two protocols, the overheads are acceptable: 38% and 43%.

The PLUG chip devotes 74% of the area to memories and the remaining 26% to computation cores and routers. Column 5 in Table 5 shows the area occupied by a PLUG chip if sized to match the needs of the protocol alone. If a protocol required only 4 tiles, then we report the area of a 4-tile PLUG in this column. We are not counting the area of unused tiles as overhead because they can be used by other lookup objects. Column 6 shows an aggressive estimate of the minimum area required for a specialized lookup module. For this aggressive estimate, we assumed no area for any of the processing required and count only the area of memories assuming no area losses due to alignment problems when laying out the memories corresponding to the logical pages. IPv4 for example would require eight individual SRAMs whose sizes match the size of the logical pages listed in Table 5. We notice that the area overheads, introduced due to the generality of PLUG can be as high as 88%. However, this comparison is to an idealized implementation. The difference will be smaller when comparing PLUGs to realistic specialized implementations.

Metric	PLUG	NLA9000
Area (mm^2)	140	-
Power (Watts)	1.4	6.5
Throughput (billion lookups/second)	633	300
Latency of lookup (ns)	148	160

Table 6: PLUG comparison to NetLogic NLA9000 (both at 55nm technology, interface overheads ignored for PLUG).

Protocol	PLUG	Ideal
Ethernet	0.72	0.22
IPv4	1.06	0.22
Seattle	1.68	0.33
Ethane	1.36	0.32

Table 7: Worst case power (Watts).

Power: Table 7 shows the power consumed by the PLUG for different protocols derived using our power model by determining the activity number of each of the protocols. Columns 2 shows total power and column 5 shows the power consumed by an ideal implementation which requires no power for processing and consumes only memory read/write power. We see that PLUGs are within 5X of this oracle implementation that simply cannot be physically constructed.

Comparison to state-of-the-art: The NLA9000 from NetLogic is a chip widely used for IP lookups and it can be configured as TCAM or as a low-power SRAM-based IP lookup engine using a proprietary lookup algorithm. It can fit 1.5 million IPv4 prefixes, it does 300 million lookups per second at a latency of 160ns consuming 6.5 Watts. Since this chip is built at 55nm technology we compare to a PLUG chip also at 55nm technology for this discussion. Because the SRAMs are slower, the highest frequency for the PLUG is 633 MHz which reduces throughput. We can accommodate 1.5 million IPv4 prefixes with a 140 mm^2 PLUG providing 9 MB of storage using 36 tiles. In this configuration the PLUG would consume 1.4 Watts and provide a latency of 148ns. Table 6 shows a summary of this comparison. Our numbers for PLUG do not account for the power and latency of the interfaces to the rest of the system, but the numbers for NLA9000 include these interface overheads also. We conclude that in this case the PLUG is actually more efficient than a specialized lookup chip.

8. RELATED WORK

Lookup modules: Ternary content addressable memories (TCAMs) use hardware parallelism to match the search key against all entries. They have been used for IP lookup [38, 42]. Their main problem is their large power consumption due in large part to the fact that all entries are activated in parallel. While selective activation of TCAM blocks reduces the power consumption [54], SRAM-based algorithmic lookup modules are the preferred solution for large forwarding tables. Pipelined tries [8, 17, 6, 32, 27] are used by algorithmic lookup modules. The PLUG solution for IP lookup falls into this category. Casado et al. [16] propose a flexible lookup module that can accommodate protocol changes. Their design is based on a TCAM cache that hands packets over to software when it cannot make a forwarding decision. Unlike PLUG which can deliver predictable throughput irrespective on the traffic mix, this solution depends on a high cache hit rate to achieve good performance.

Dataflow: The PLUG programming model is inspired originally by the SDF model [36] and by the dataflow model of Click [30]. The core differences are that we focus on lookups, not on the entire functionality of the router and that our goal is to map the objects to a module specialized for lookups, not to a general-purpose processor. Gordon et al. discuss a general dataflow like approach called StreamIt [24]. Similar to historical dataflow machines [21, 4] the PLUG architecture implements dataflow execution but in a coarse granularity of code blocks and network messages. The conflict-free operation of PLUG pipelines is similar to systolic arrays that execute in a very regular fashion.

Tiled architectures: The PLUG architecture is inspired by recently proposed tiled architectures [46, 37, 41, 45]. The key distinction is that these architectures are targeting general-purpose processing and thus include area- and power-consuming features such as memory disambiguation, control speculation, networks with flow control and buffering and are too inefficient for workloads dominated by memory accesses such as the lookups. Targeting lookups allows us to statically eliminate all resource conflicts in PLUG. Thus our architecture spends less area and power on "overhead" beyond the memories required to store the forwarding table (internal communication and processing) and achieves higher throughput for lookups than existing tiled architectures because we can fully pipeline processing in each tile. The suitability of tiled architectures to network-related tasks has been investigated in [40]. The authors developed a router architecture and deployed it on the MIT RAW tiled microprocessor. The goal of this work is slightly different from ours, as it aims to implement the full routing process, while the PLUG is specialized to provide lookup support for an external processing element. Moreover, the applicability of the RAW-based approach to tasks other than IP routing has not been demonstrated.

9. CONCLUSIONS AND FUTURE WORK

High speed network processors use custom modules for lookups in forwarding tables because of area, power, and performance advantages. Since most new protocols require different structures for their forwarding tables, inflexible lookup modules customized to current protocols slow down the deployment of new ones by making hardware upgrades necessary. We propose PLUG, an architecture for a flexible lookup module and we show how the forwarding tables of four different protocols can be mapped to it.

The forwarding tables are expressed as lookup objects that break the data structure into logical pages, break the processing into code blocks each local to one page and use explicit messages for all data transfers. The dataflow graph describes the internal communication patterns of the lookup object. The data, processing, and communication of the lookup object are mapped to the tiled PLUG architecture in a way that avoids resource conflicts producing a pipeline with a fixed throughput of one lookup per cycle. Each tile has multiple $\mu core$s to process one new request every cycle. The static avoidance of resource conflicts makes the routers used for internal communication and the $\mu core$s very simple.

We have shown that PLUGs compare favorably to other approaches. Future work to refine the architecture, compiler, and system software is required to definitely answer the question of how effective PLUGs are in deployed systems. Further evaluation in a product environment can best answer some of the questions that relate to the economics of high-end routers.

Acknowledgments

This work is sponsored by NSF grants 0546585, 0627102, and 0716538 and by a gift from the Cisco University Research Program Fund at Silicon Valley Community Foundation. We thank Pere Monclus, Mike Swift, Mike Ichiriu, Randy Smith, Matt Fredrikson and the anonymous reviewers for suggestions that improved this paper.

10. REFERENCES

[1] Semiconductor Industry Association (SIA), Process Integration, Devices, and Structures, International Roadmap for Semiconductors, 2005 edition. Int. SEMATECH, 2005.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, pages 63–74, Aug. 2008.

[3] D. G. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker. Accountable internet protocol (AIP). In *Proceedings of the ACM SIGCOMM*, Aug. 2008.

[4] Arvind and D. E. Culler. Dataflow Architectures. *Annual Review of Computer Science*, 1:225–253, 1986.

[5] F. Baboescu, S. Singh, and G. Varghese. Packet classification for core routers: Is there an alternative to CAMs. In *INFOCOM*, Apr. 2003.

[6] F. Baboescu, D. Tullsen, G. Rosu, and S. Singh. A tree based router search engine architecture with single port memories. In *ISCA*, June 2005.

[7] F. Baboescu and G. Varghese. Scalable packet classification. In *SIGCOMM*, pages 199–210, Aug. 2001.

[8] A. Basu and G. Narlikar. Fast incremental updates for pipelined forwarding engines. In *INFOCOM*, Apr. 2003.

[9] M. Becchi and P. Crowley. An improved algorithm to accelerate regular expression evaluation. In *ANCS*, Dec. 2007.

[10] M. Becchi and P. Crowley. Efficient regular expression evaluation: Theory to practice. In *ANCS*, December 2008.

[11] N. Beheshti, Y. Ganjali, M. Ghobadi, N. McKeown, and G. Salmon. Experimental study of router buffer sizing. In *Internet Measurement Conference*, Oct. 2008.

[12] F. Bonomi, M. Mitzenmacher, R. Panigraphy, S. Singh, and G. Varghese. Beyond Bloom filters: From approximate membership checks to approximate state machines. In *SIGCOMM*, Sept. 2006.

[13] A. Broder and M. Mitzenmacher. Using multiple hash functions to improve IP lookups. In *INFOCOM*, pages 1454–1463, Apr. 2001.

[14] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: taking control of the enterprise. In *SIGCOMM*, Aug. 2007.

[15] M. Casado, T. Garfinkel, A. Akella, M. Freedman, D. Boneh, N. McKeown, and S. Shenker. SANE: A protection architecture for enterprise networks. In *USENIX Security Symposium*, Aug. 2006.

[16] M. Casado, T. Koponen, D. Moon, and S. Shenker. Rethinking packet forwarding hardware. In *HotNets-VII*, Oct. 2008.

[17] F. Chung, R. Graham, and G. Varghese. Parallelism versus memory allocation in pipelined router forwarding engines. In *SPAA*, pages 103–111, June 2004.

[18] Cisco Public Information. The cisco quantumflow processor: Cisco's next generation network processor. http://www.cisco.com/en/US/prod/collateral/routers/ps9343/solution_overview_c22-448936.html, 2008.

[19] D. Clark, J. Wroclawski, K. Sollins, and R. Braden. Tussle in cyberspace: Defining tomorrow's internet. In *SIGCOMM*, August 2002.

[20] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. In *SIGCOMM*, pages 3–14, Oct. 1997.

[21] J. Dennis. A preliminary architecture for a basic data-flow processor. In *ISCA '75*, pages 126–132, January 1975.

[22] W. Eatherton. The push of network processing to the top of the pyramid. Keynote Address at ANCS, Oct. 2005.

[23] P. Francis and R. Gummadi. IPNL: A NAT-extended internet architecture. In *SIGCOMM*, pages 69–80, Aug. 2001.

[24] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, C. Leger, A. A. Lamb, J. Wong, H. Hoffman, D. Z. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *ASPLOS 2002*, San Jose, CA USA, Oct. 2002.

[25] C. Guo, H. Wu, K. Tan, L. Shiy, Y. Zhang, and S. Luz. DCell: A scalable and fault-tolerant network structure for data centers. In *SIGCOMM*, pages 75–86, Aug. 2008.

[26] P. Gupta and N. McKeown. Packet classification using hierarchical intelligent cuttings. In *Hot Interconnects VII*, Aug. 1999.

[27] W. Jiang, Q. Wang, and V. K. Prasanna. Beyond TCAMs: An SRAM-based parallel multi-pipeline architecture for terabit IP lookup. In *INFOCOM*, Apr. 2008.

[28] D. A. Joseph, A. Tavakoli, and I. Stoica. A policy-aware switching layer for data centers. In *SIGCOMM*, pages 51–62, Aug. 2008.

[29] C. Kim, M. Caesar, and J. Rexford. Floodless in seattle: A scalable ethernet architecture for large enterprises. In *SIGCOMM*, pages 3–14, Aug. 2008.

[30] E. Kohler, R. Morris, B. Chen, J. Jannotti, and F. Kaashoek. The Click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, Aug. 2000.

[31] S. Kong, R. Smith, and C. Estan. Efficient signature matching with multiple alphabet compression tables. In *SecureComm*, Sept. 2008.

[32] S. Kumar, M. Becchi, P. Crowley, and J. Turner. CAMP: fast and efficient IP lookup architecture. In *ANCS*, pages 51–60, Dec. 2006.

[33] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *SIGCOMM*, Sept. 2006.

[34] S. Kumar, J. Turner, and J. Williams. Advanced algorithms for fast and scalable deep packet inspection. In *ANCS 2006*, pages 81–92.

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

[36] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[37] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart Memories: A modular reconfigurable architecture. In *ISCA*, pages 161–171, June 2000.

[38] A. J. McAuley and P. Francis. Fast routing table lookup using CAMs. In *INFOCOM*, pages 1382–1391, Apr. 1993.

[39] N. McKeown. The NetFPGA project. http://www.netfpga.org/.

[40] U. Saif, J. W. Anderson, A. Degangi, and A. Agarwal. Gigabit routing on a software-exposed tiled-microprocessor. In *ANCS*, pages 51–60, Oct. 2005.

[41] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, S. W. Keckler, D. Burger, and C. R. Moore. Exploiting ILP, TLP and DLP with the Polymorphous TRIPS Architecture. In *ISCA '03*, pages 422–433, June 2003.

[42] D. Shah and P. Gupta. Fast updating algorithms for TCAMs. *IEEE Micro*, 21(1):36–47, Jan. 2001.

[43] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. In *SIGCOMM*, 2003.

[44] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. In *SIGCOMM*, Aug. 2002.

[45] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. Wavescalar. In *MICRO '03*, pages 291–302, December 2003.

[46] M. B. Taylor, J. Kim, J. Miller, D. W. laff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, W. L. Jae-Wook Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The RAW Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro*, 22(2):25–35, March 2002.

[47] Xtensa lx2: The fastest processor core ever, http://www.tensilica.com/products/xtensa_lx.htm.

[48] S. Thoziyoor, N. Muralimanohar, and N. Jouppi. Cacti 5.0. Technical Report HPL-2007-167, HP Research Labs, 2007.

[49] B. Vöcking. How asymmetry helps load balancing. In *IEEE-FOCS*, pages 131–140, Oct. 1999.

[50] H.-S. Wang, X. Zhu, L.-S. Peh, and S. Malik. Orion: a power-performance simulator for interconnection networks. In *MICRO 35*, pages 294–305, 2002.

[51] A. Yaar, A. Perrig, and D. Song. SIFF: A stateless internet flow filter to mitigate DDoS flooding attacks. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2004.

[52] X. Yang, D. Clark, and A. W. Berger. NIRA: a new inter-domain routing architecture. *IEEE/ACM Transactions on Networking*, 15(4):775–788, Aug. 2007.

[53] X. Yang, D. Wetherall, and T. Anderson. A DoS-limiting network architecture. In *SIGCOMM*, Aug. 2005.

[54] F. Zane, G. Narlikar, and A. Basu. CoolCAMs: Power-efficient TCAMs for forwarding engines. In *INFOCOM*, Apr. 2003.