Experiences in Co-designing a Packet Classification Algorithm and a Flexible Hardware Platform

Nilay Vaish† Thawan Kooburat† Lorenzo De Carli†
Karthikeyan Sankaralingam† Cristian Estan‡
†University of Wisconsin-Madison ‡NetLogic Microsystems
nilay@cs.wisc.edu kooburat@cs.wisc.edu lorenzo@cs.wisc.edu karu@cs.wisc.edu estan@netlogicmicro.com

ABSTRACT

Algorithmic solutions to the packet classification problem in network equipment have long been a subject of study in academia and industry and with increases in network speeds they are becoming even more important. Since general purpose processors cannot meet performance and cost requirements, researchers have been assuming that ASICs or FPGAs are necessary for hardware implementation.

Industry and academia have been working on SRAM-based platforms specialized for tables used in network equipment, but existing publications only describe the mapping of simpler exact match or prefix match lookups to such platforms. In this paper we adopt a software-hardware co-design approach mapping the EffiCuts algorithm to the PLUG platform. Our work confirms that this solution achieves high throughput (142 million packets per second) and low power (3.1 Watts). It identifies and evaluates changes to the original algorithm and to the platform that can improve throughput and memory utilization.

Categories and Subject Descriptors: B.4.1 [Data Communication Devices] Processors; C.1 [Computer Systems Organization]System Architectures; C.2.2 [Network Protocols]Protocol Architecture (OSI model)

General Terms: Design, Algorithms, Performance

Keywords: Packet Classification, Network processing, Lookups, TCAM

1. INTRODUCTION

The processing of each packet in routers and switches involves matching fields from the packet's headers against multiple tables to decide how to treat it (the direction towards the destination, the level of service, conformance with security policies, etc.). The type of data in such tables (MAC addresses, IP prefixes, access control lists, etc.) determines the type of matching operation the network equipment must implement. Packet classification is the most challenging type of table matching operation.

In packet classification, entries are matched against multiple fields of the packet headers and each entry can specify a mix of exact values, wildcards, prefixes and ranges that must match. Such matching of packets against tables at line rates is a central challenge for the architecture of routers and switches. In the past, DRAM-based solutions were feasible and attractive because of the low cost of commodity DRAM. For current and future line rates that are hundreds of gigabits per second, such solutions are infeasible. This is because of the low random read rate of commodity DRAMs and the bad temporal locality of the memory accesses. This bad locality occurs because the traffic stream mixes packets of many flows, with little overlap in the sets of table entries matched by the packets of different flows.

Industry's answer to this challenge is the adoption of architectures storing the tables in fixed locations and integrating table processing and storage onto the same chip. An example of one such architecture is ternary content addressable memories (TCAMs) which have been widely adopted by industry for packet classification. By using extreme hardware parallelism (one comparator for each ternary bit), today's TCAMs achieve packet processing rates required by interfaces running at hundreds of gigabits per second.

Despite TCAMs' success, academia and industry alike have sustained a long-lasting line of work on algorithmic solutions to the packet classification problem [2, 12, 7, 15, 19]. Such solutions can use SRAMs to store the tables giving them two potential advantages over basic TCAMs: lower power and higher density. Compared to DRAMs, SRAM solutions provide lower latency and can sustain higher bandwidth. A recurring problem with various algorithms is that the data structures they need to build can become orders of magnitude larger than the original table, negating their appeal. Hence much of the academic effort has been on improving algorithms to limit or eliminate the memory overhead as compared to raw tables with recent algorithms such as EffiCuts [19] achieving good results.

The research on algorithms for packet classification has been largely separated from the work on platforms to deploy these algorithms on, as authors typically assumed deployment on FPGAs or ASICs implementing the algorithm. FPGAs have the problem of high power consumption and low density as SRAMs are a relatively small fraction of the chip area (between 2% to 15% depending on the FPGA[1]). Hard-coding algorithms in ASICs has the problem of low flexibility which makes the chip volumes lower as individual ASICs can satisfy only small sub-segments of the overall market for high-speed lookups in tables used by network equipment.

An alternative direction is to develop flexible *lookup processors*. These are programmable SRAM-based platforms that can perform various types of lookups in a wide variety of tables. Such flexible lookup processors can then be deployed on line-cards as standalone chips or as modules within network processors. This direction has been adopted by both proprietary technologies within network equipment manufacturers and academic proposals such as the PLUG system [4, 11]. The PLUG platform has high memory band-

[1]We acknowledge this is hard to quantify accurately as FPGA die-sizes are not public. Our estimate is based on considering the largest of Xilinx FPGAs. XC5VFX200T at 65nm allows about 2.5 MB total of SRAM (with BRAM and distributed RAM). It has, mostly likely, the largest possible die-size in the $25mm \times 25mm$ range. Using CACTI or other tools, the area of an equivalent 2.5MB dedicated SRAM is 50 mm^2 at 65nm. Comparing these two die areas, we can estimate that SRAM is 8% of the FPGA.

width, the flexibility required to accommodate packet classification algorithms, and its SRAM density exceeds 70%. Initial workloads for PLUG have focused on tables requiring the simpler exact match and longest prefix match lookups.

This paper investigates an implementation of the hardest matching operation – packet classification – on the PLUG architecture. Specifically, we implement a recent packet classification algorithm – EffiCuts – on PLUG. Contrasting with the algorithm-centric work, we adopt a co-design approach. This co-designed implementation involves three major steps.

- Implementing the EffiCuts algorithm in PLUG's dataflow graph programming model (Section 3)
- Modifying and extending PLUG's execution rules, compiler, and hardware such that this complex application can be mapped. (Section 4)
- Evaluation of our final solution to understand how well the architecture is used and comparison to the state-of-art in packet classification. (Section 5)

Our co-designed approach and analysis confirms that the PLUG architecture and its dataflow-graph-based programming model are flexible enough to easily accommodate a new complex lookup algorithm such as EffiCuts. But we go much beyond the original mapping. We explore changes to both the original algorithm and the original platform that make the combination of the two a more compelling solution for packet classification. We believe our findings can also inform the design of other algorithms and architectures targeting high throughput. The three main findings are:

1. The initially proposed simple heuristics for mapping data structures to the memories of the PLUG tiles cannot achieve good memory utilization. We found efficient solutions by formulating the memory mapping as an integer linear programming problem and using off-the shelf tools to solve it.
2. Constraining the node sizes in the EffiCuts data structure to powers of two eliminates the need for expensive integer divisions reducing latency by 30% (from $1.37\mu s$ to $0.95\mu s$) with little effect on memory requirements.
3. Significant internal fragmentation problems arise for memories because of PLUG's fixed bank-size leading to underutilization of the memories.

The rest of the paper is organized as follows. Section 2 provides background and discusses the EffiCuts algorithm and the PLUG architecture. Sections 3 and 4 describe our co-designed implementation. Section 5 presents our evaluation results, Section 6 discusses lessons learned, Section 7 discusses related work. Finally, Section 8 concludes.

2. BACKGROUND

2.1 Packet Classification Using Decision Trees

Packet classification rules contain ranges for five different fields: source IP, destination IP, source port, destination port and protocol. Classifying packets by iterating through the entire set of rules is too slow, therefore decision-tree based solutions are common. Each rule can be viewed as a 5-dimensional hypercube that may overlap with other rules. A decision tree is constructed by dividing the rule space into a hierarchy of regions. The root of the tree represents the entire rule space. This space is divided amongst its children, which are further subdivided. The subdivision process is carried out until the number of rules contained in each region is small enough. Figure 1 shows an example decision tree

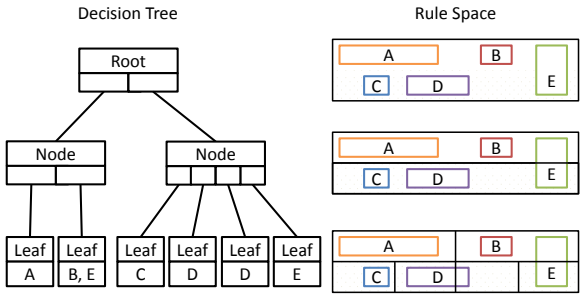

Figure 1: An Example Decision Tree

Each packet is a point in the rule space. Classifying a packet means finding the region to which this point belongs and reporting the highest priority matching rule. As the tree is traversed from top to bottom, the region to which this packet belongs is narrowed down. At the leaf node, the packet is matched against the rules within the region associated with the leaf node.

Gupta and McKeown introduced the idea of using a decision tree for packet classification [7]. Singh et al. improved upon this technique to reduce the memory space required [15]. Other optimizations have been explored in the literature and in general on the order of 1 GB of space is required for storing 100K rules.

2.2 State-of-the-art Algorithm: EffiCuts

Vamanan et al. have recently introduced an algorithm named EffiCuts based on two observations [19]. Their first observation is that, as the rule space is divided into a hierarchy of regions, it is very likely that rules span multiple regions and hence are replicated amongst leaf nodes. EffiCuts avoids this problem by partitioning the rule set. Rules with different patterns of wildcards are assigned to different subsets and each subset has its own separate decision tree. This optimization cuts down rule replication and reduces the total memory required.

Their second observation is that rule distribution is uneven over the rule space. Due to this unevenness, a large number of nodes representing empty regions are created which unnecessarily increase the size of the decision tree. To mitigate this problem, EffiCuts fuses such nodes together into a smaller set of nodes. In particular, EffiCuts use two types of intermediate nodes: *equi-size* and *equi-dense*. As shown in Figure 2 (a), an equi-size node divides its rule space equally amongst its children. On the other hand, an equi-dense node, which is shown Figure 2 (b), tries to assign the same number of rules amongst its children.

A packet is classified by traversing the decision tree and performing lookup operations on nodes along the path. For each type of node, the first step is to calculate the position of the packet relative to the region associated with the node. This relative position is translated into an *offset* which is used to locate the child to be visited next. Since an equi-size node divides its region equally among its children, the offset can be used to index directly into the array storing the child pointers. In the case of an equi-dense node, every child's region size is different. An *interval* array stores the boundary of each child's region. Therefore, a linear scan through the interval array is required to locate the corresponding child.

Figure 2 shows an example of the lookup process. In this example, the incoming packet is represented as a point (2,1) relative

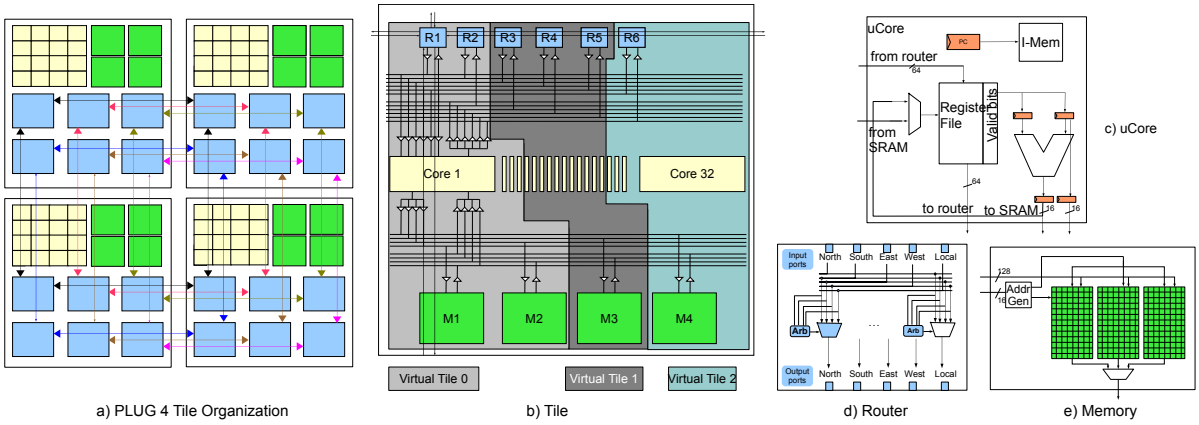

Figure 3: PLUG Overview

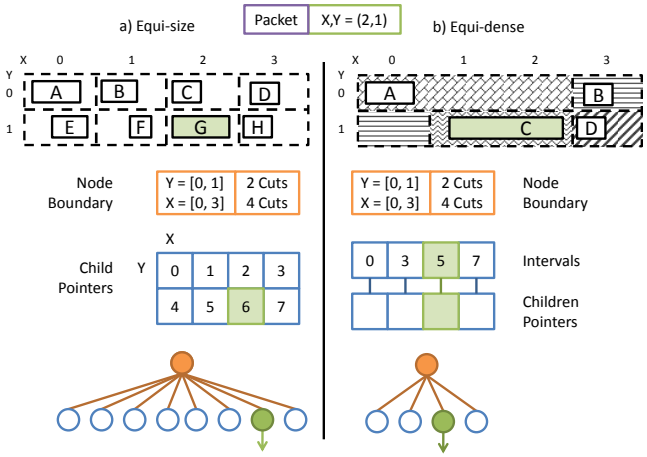

Figure 2: Types of Nodes in EffiCuts

to current node's region. For the equi-size node, the next node to be looked up is the 7^{th} child. For the equi-dense node, scanning through interval array identifies the 3^{rd} child as the next node.

Vamanan et al. [19] focused on algorithmic evaluation. They abstracted away hardware implementation details and implicitly assumed a specialized ASIC that controls SRAM.

2.3 Pipelined Lookup Grid: PLUG

PLUG is a recently proposed lookup processor with a novel tiled architecture, dataflow based programming model, and a compiler toolchain that maps lookup algorithms written as C++ programs to the hardware. Its goal is to achieve low-power and performance like an ASIC while being easily programmable. We describe its salient features below, while details are in other papers [4, 11].

Programming model: The PLUG architecture is programmed using the data flow programming model. The algorithm associated with the lookup operation is divided into steps which are linked together to form a data flow graph. Each node in the graph represents a portion of the data structure being looked up. The edges of the graph represent the communication pattern. In PLUG terminology, a node in the data flow graph is referred to as a *logical page*. A logical page contains *data* and *code-blocks*. The code-blocks are the methods acting on the data as part of the lookup operation. Messages, which are sent along the edges of the graph, are used for communication between the code-blocks. The execution of a code-block starts when the logical page receives a message which also indicates which code-block to execute.

Hardware Architecture: PLUG is a tiled hardware architecture. Each tile has a set of in-order μcores, SRAM banks, and routers for communicating with other tiles. An overview of the architecture is shown in Figure 3.

Task scheduling: Compared to a conventional multicore processors, where different cores process different packets in parallel, PLUG tiles are used in a pipelined fashion. Different steps of a packet processing task are mapped to different tiles, and packet information flows from one tile to the next one as steps are completed. Logical pages are broken into physical pages that can fit within the local memory available in a tile.

A consequence of this execution model is that the processing performed by each tile is expected to be both simple and short. This allows the architecture to be statically scheduled, which in turn simplifies processor design and reduces power consumption. There is no arbitration when multiple cores access the network or the memory banks. Hence, programs must statically guarantee that their accesses to resources are contention-free. These guarantees are referred to as the execution rules of the architecture and they allow the architecture to sustain a guaranteed throughput of one lookup every cycle.

Execution Rules:

1. Due to static scheduling, all code-blocks in a tile must execute the same number of instructions. Each execution flow can go through a different path in the data flow graph, but the execution time for each path should be the same.

2. To have contention-free resource access, instructions in all code-blocks can access networks or memory banks only at the same number of instructions (cycles) from the start of the code-block. This allows cores to interleave their accesses with one another.

3. Only one memory access is allowed per code-block to so we can statically ensure that there will be no conflicts for accessing memories.

4. Code-blocks cannot have more than 32 dynamically executed instructions.

5. The scheduling of code-blocks' network instructions and the mapping of the physical pages to tiles must statically ensure that there is no contention amongst messages arriving at a tile.

Performance and Capabilities: With a clock frequency of 1 GHz, the PLUG processor can sustain a throughput of a billion decisions per second provided that all the above rules can be enforced. Considering 40-byte packets, PLUG can therefore sustain 320 Gbps throughput on packet classification. In case of EffiCuts, the complexity of the algorithm requires relaxation of some constraints, which in turns leads to a throughput of less than one decision per cycle.

In the next two sections, we describe how the EffiCuts algorithm is implemented on the PLUG programming model and then discuss algorithm and architecture extensions.

3. ALG. & PROG. MODEL CO-DESIGN

In this section we describe our design of EffiCuts for the PLUG's dataflow programming model.

3.1 Decision Tree Construction

One of the insights of EffiCuts is to generate multiple decision trees to achieve memory reduction. Each level in a decision tree forms a logical page. Messages between the decision tree's logical pages encapsulate all the information required to decide on which child node to access in the next level. The data portion of the logical page consists of the nodes occurring at that level. The code-blocks implement the processing required to determine the next child. The logical data flow graph of a single decision tree is a linear sequence of logical pages as shown in Figure 4b. Multiple decision trees are combined with an *input node* at the head and an *output node* at the tail to form the full dataflow graph. Figure 5 shows an actual example of five decision trees combined together into a physical data flow graph.

We now describe the details of the data contained in the nodes. We had to carefully consider what is allowed in the programming model to successfully map the algorithm. Each non-leaf node consists of three fixed-size sections – (i) *header* , (ii) *boundary*, and (iii) *interval*. Table 1 describes the contents of these sections. The node also contains pointers to the children present in the next level. The number of children of a node is dependent on the node itself and hence its size is not fixed. To simplify the lookup process, the logical page's memory layout is as follows. The fixed portions are first laid out contiguously. Then, we layout the children pointers (the addresses for children in the next logical page) for each node.

All the leaves in the tree are stored in a separate logical page. We made this choice to simplify the design. A leaf node consists of two sections – a header and a fixed length array of indices of rules associated with the leaf. The rules are stored contiguously in the rule array after all the leaf nodes. Table 1 also shows how leaf nodes are laid out.

3.2 Lookup Process

The lookup operation starts when a message containing the packet header arrives at the 1st level logical page of a decision tree. In this logical page there is only one node to lookup. On other non-leaf logical pages, the index of the node to be looked up is extracted from the message and is used to load the node's header, boundary and interval sections from the memory. This requires three memory accesses. Then, the index of the child to be looked up next is calculated using the packet header and boundary of the region associated with the node. After that, one more memory access is required to read address of the child which is to be looked up next. This address and the packet header are sent to the next logical page. However, one of PLUG's execution rules requires all executions paths to be equal. Packets which are rejected early in the lookup process need to be forwarded through all the levels in order to guarantee that every packet is processed in equal amount of time .

When the packet reaches the leaf logical page, the leaf header and rule IDs are read. Then, the packet is matched against every rule associated with this leaf node. Finally, the index of the highest priority rule that matched the packet or no-match is returned as the final result. The *output node* combines matching results from all decision trees.

4. IMPLEMENTATION, COMPILER, HARDWARE CO-DESIGN

Previously we described how EffiCuts algorithm is mapped to the PLUG programming model. We now describe implementation of the algorithm which had to be co-designed with the execution rules in mind. To this end, we describe the implementation of the dataflow graph and the code-block. We then describe extensions to the compiler and hardware to map the implementation. While the original PLUG compiler and hardware were too rigid to allow the mapping of EffiCuts, some simple extensions were sufficient.

4.1 Implementing the Logical Dataflow Graph

The abstract programming model specifies pages at arbitrary sizes. To implement this on the PLUG architecture, the pages must be converted into so-called physical pages. Since each PLUG tile has limited amount of memory, we need to break the logical pages into physical pages small enough to fit within a single tile. After such a breakup, we obtain what is called the physical data flow graph. This transformation is shown in Figures 4b-c and an example for ACL with 100K rules is in Figure 5. Unlike the *simple* matching applications that were previously investigated, automatically generating the physical pages from the logical pages proved intractable. This is because the nodes contain information on the *next* page. On the logical dataflow graph this is just a label, but on the physical dataflow graph, this must be converted into a tile coordinate. Our current software toolchain and compiler does not perform this step automatically. Hence, we used a two-step process. We directly specified the physical dataflow graph by breaking pages into optimal sizes. However, at programming time, we do not know where pages will be mapped. After scheduling has been done, the contents of the nodes are re-written according to the schedule.

4.2 Implementing the Code-blocks Removing Division Operations

Implementing the code-blocks was straightforward, but the original algorithm results in extremely long code-blocks. On detailed investigation, we observed that the original EffiCuts algorithm required the use of division operations during the lookup process. The code below illustrates the need for division operation –

```
cut_size = (boundary.source_ip_high -
    boundary.source_ip_low + 1) / num_cuts;
dim_index = (pkt->source_ip - boundary.
    source_ip_low) / cut_size;
```

Listing 1: Code showing use of Division Operation

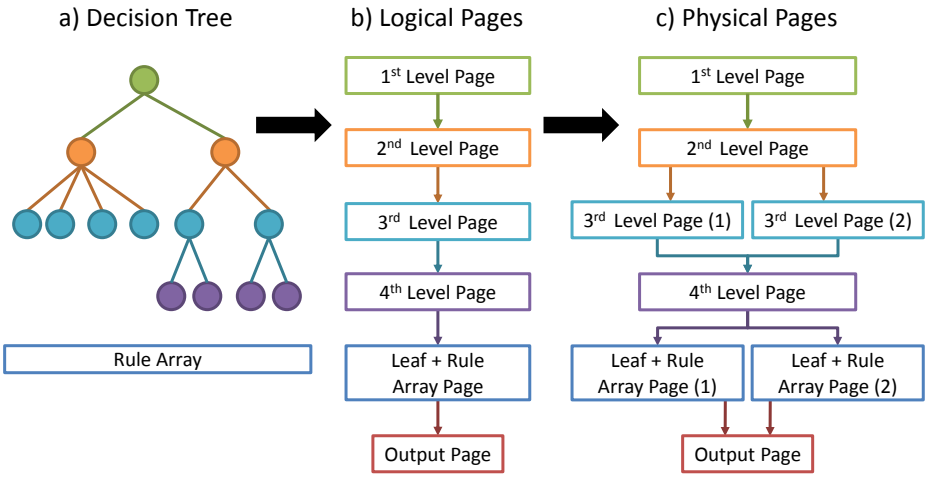

Figure 4: Mapping EffiCuts to PLUG

Data Structure	Size (bytes)	Content		
		Equi-size Internal Node	Equi-dense Internal Node	Leaf
Header	4	Type, Interval count, cuts, dimension		node type, number of rules
Boundary	28	Range of each dimension		-
Interval	16	Pointer to child pointers	7 intervals, Pointer to child pointers	Rule IDs
Child Pointers	4 per pointer	Up to 1024 pointers	Up to 7 pointers	-

Table 1: Node Format

The size of a cut is calculated by subtracting the high and the low boundary values and then dividing with the number of cuts. The number of cuts is a power of two, so this division can be done using shifts. But in the next line, the index of the child that needs to be looked up next is computed. But this requires division by the cut size, which may not be a power of two. Hence, the need for the division operation. There were two places in our implementation of the packet classifier where division operation was required.

But division operations are costly in terms of number of cycles required per operation. In fact, the two division operations accounted for about $1/3$ of the total cycles required for execution of a non-leaf code-block. Hence, we decided to eliminate these division operations converting them to shift operations. The starting range of any field (IP, port, protocol) is a power of two. As the number of cuts being made over the range is always a power of two, so even the children node will always have a range which is a power of two. The original EffiCuts algorithm tries to squeeze this range by looking at the rules themselves, so as to optimize on the memory space required. After the range of a node is adjusted, we expand this range so that it is a power of two. Depending on the dataset, this results in an increase in memory from 0 to 12%. This co-designed algorithm/implementation optimization results in about 30% reduction in code-block length (from 200 cycles to 130 cycles). Depending on the dataset, this can translate to total latency reduction from $1.37\mu s$ to $0.95\mu s$.

4.3 Extending the Execution Rules

To ensure contention-free execution within a tile, the current PLUG architecture imposes several restrictions on the code-block. As described in Section 2.3, these are referred to as the execution rules (ER). The two relevant ones here are: ER-3 - a code-block can make at most one memory access, and ER-4 - code-blocks can be at most 32 instructions long. *With these execution rules, packet classification cannot be implemented on PLUG.* Even the division optimization did not get instruction counts to under 32.

Examining the architecture's constraints, we observed that these rules are more strict than necessary and reduce PLUG's ability to support applications like packet classification that need many accesses to memory. In this section we describe our extension for allowing arbitrary number of memory accesses in a code-block and arbitrarily long code-blocks.

First, we present a method called *Modulo Interleaving* that the PLUG compiler can adopt to ensure contention-free execution for these large code-blocks that may make more than one memory access. Note that these memory accesses can appear anywhere in the code-block. We then discuss the relaxing of code-block length.

Modulo Interleaving: Consider the example shown in Figure 6a. There are three memory accesses in cycles X, Y and Z. A naive way of ensuring contention-free execution would be to separate two successive executions of the code-block by enough number of cycles so that by the time a code-block execution accesses the memory for the first time, the previous execution has already completed all the required memory accesses. In the given example, we can separate the two executions by at least Z-X+1 cycles to ensure that the two memory accesses never overlap; however, this is far from optimal.

In modulo interleaving, we intelligently interleave memory accesses when they happen in an irregular fashion across the code-block. The compiler generates instructions for a code-block in a manner such that memory accesses from different executions of

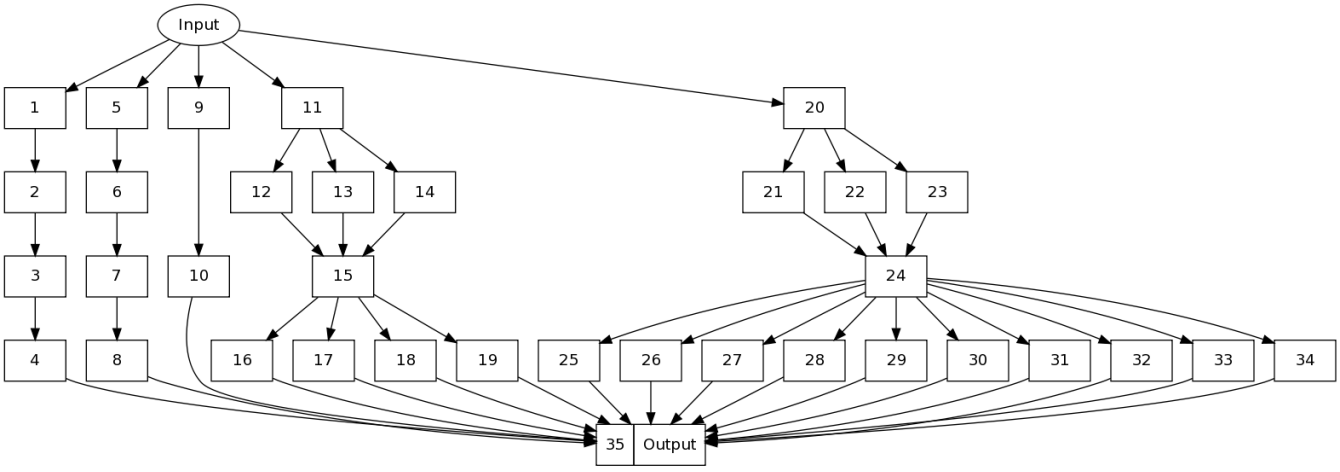

Figure 5: Physical Data Flow Graph for ACL 100k rule set

the code-block never contend. The compiler can ensure this by *reordering* instructions and/or by adding *no-op* instructions. We provide two conditions which the compiler can impose on the code-block to ensure contention free execution. Assume, the total number of memory accesses within a code block is m and that the instruction slots occupied by the memory accesses are $s_1, s_2, \cdots, s_m$. Then the conditions are:

1. Two different executions of the code-block are separated by a multiple of m, the number of memory accesses.
2. $s_i \mod m \neq s_j \mod m$ for all $i \neq j$.

With this approach, more than one memory access can be allowed but the throughput reduces from one lookup every cycle to $1/m$ lookups every cycle. Next, we prove that these conditions ensure contention-free execution of code-blocks.

PROOF. Consider two executions of a code-block on a PLUG tile. We refer to the first execution as A and the second execution as B. We assume that instruction slots for memory accesses have been scheduled as per the conditions listed above and that these code-blocks are separated by $k * m$ cycles, where m is number of memory accesses in the code-block and k is any natural number. Suppose A is executing the i^{th} instruction and B is executing the j^{th} instruction. Then, using the fact that the separation between code-blocks is $k * m$, we get

$$\begin{aligned} & i && = j + k * m \\ \Rightarrow & i \mod m && = (j + k * m) \mod m \\ \Rightarrow & i \mod m && = j \mod m \end{aligned}$$

Since $i \equiv j \mod m$, therefore the second condition forces that at most one of these is a memory access. Hence, the memory accesses are contention-free. □

In our example in Figure 6a, $m = 3$, $s_1 = X$, $s_2 = Y$ and $s_3 = Z$. To ensure contention-free execution, the first condition states that any two executions of this code block need to separated by three cycles. The second condition states that $s_i \mod 3 \neq s_j \mod 3$. The assignment $X = 0, Y = 2$ and $Z = 7$ satisfies this condition. Hence in Figure 6b, there is no contention in memory accesses.

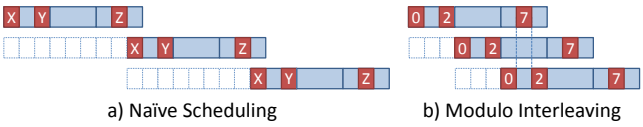

Figure 6: Modulo interleaving with $m = 3$

Code-block Length: ER-4 is enforced together with ER-3 in order to ensure that 32 μcores are enough to sustain peak throughput. Code-block length cannot exceed 32 instructions so that we have enough μcore to initiate a new lookup every cycle. The first μcore must finish executing its code-block when the 32^{nd} μcore starts execution. By relaxing ER-3 to allow m memory accesses and starting a new lookup every m cycles, we can also relax ER-4. The code-block can have a length of up to $32 * m$ instructions (cycles) while retaining $1/m$ of peak throughput. This extension requires some simple hardware extensions to PLUG as discussed in Section 4.5.

4.4 Compiler Extensions: Optimization Based Scheduler

Using the implementation strategies and the ER extensions discussed thus far, we obtain legal code-blocks that can execute on a tile of the PLUG architecture. Next, we need to schedule the physical pages on to PLUG. The greedy scheduler that is part of the PLUG compiler [11] does not scale to the size and type of complex graphs that EffiCuts produces. Manual scheduling was also infeasible and ad-hoc greedy heuristics did not work. We instead developed a solution using formal optimization frameworks.

We modeled the problem of scheduling physical pages onto PLUG as an *Integer Linear Program*. An Integer Linear Program consists of an objective function whose value is minimized (or maximized) provided certain constraints are fulfilled. The key fact to note is that each variable in the program is constrained to take only integer values. The problem is known to be NP-hard, but algorithms and tools exist that can solve practical instances very quickly.

Before describing the objective and the constraint functions for our Integer Linear Programming formulation, we introduce certain notations. We assume that the PLUG chip contains $n \times n$ tiles. We label each tile as (i, j) where i and j are the x- and y- coordinates

minimize	n
subject to	$\sum_k X_{i,j,k} \leq 1 \; \forall i,j \leq n$
	$\sum_{i,j} X_{i,j,k} = 1 \; \forall k$
	$xpos(u) - xpos(v) \leq 0$
	$ypos(u) - ypos(v) \leq 0$
where	u and v are nodes in the graph and u is parent of v
	$xpos(u)$ and $ypos(u)$ denote the x and y coordinates of the tile u is mapped to
	$xpos(output) \leq n$
	$ypos(output) \leq n$
where $output$	represents the output node in the graph

Figure 7: Integer Linear Program for Scheduling

of the tile. Note that $1 \leq i,j \leq n$. We label each node in the data flow graph with an index. Then, for each node k in the graph, and each tile (i,j), we introduce a Boolean variable $X_{i,j,k}$ which is 1 if k^{th} node occupies the (i,j) tile, 0 otherwise.

The formulation is shown in Figure 7. We model the objective function as minimizing the size of the PLUG chip required for mapping the physical pages. Since the chip is assumed to be a square, so we minimize n. We have four different types of constraints. The first constraint in Figure 7 limits to one the number of physical pages that can be scheduled on a single PLUG tile. The second constraint forces each code-block to be scheduled on some PLUG tile. Then follow two constraints that together enforce the south-east rule (i.e. all child nodes are to the south and east of parent nodes). This constraint ensures that all the paths in the data flow graph have equal lengths, thus taking care of ER-1. Note that these two constraints are present for all (parent, child) pairs. The last constraint forces the output node to have coordinates as small as possible.

We have further formulated a variant of this where more than one physical page gets mapped to the same physical tile, but the combined memory and cores required for these physical pages are capped to the total memory and cores that a tile supports. We developed this variant to increase the utilization of the tiles and reduce the size of the chip required. While reporting the results in this paper, we will use the scheduling obtained from this variant of the originally stated formulation.

Network Assignment: In the PLUG architecture, each edge in the physical data flow graph needs to be statically mapped to one of the on-chip networks. Note that the edges which do not overlap (i.e. that do not have any common point-to-point link between tiles) can reuse the same network. We manually applied graph coloring to our data sets to obtain network assignments for them. None of the members in our data sets required more than the 6 networks provided in PLUG. This part of the compiler currently requires manual intervention and automating the procedure for network assignment is on-going work. For the rest of the paper, we assume that we have enough networks available to ensure contention-free communication.

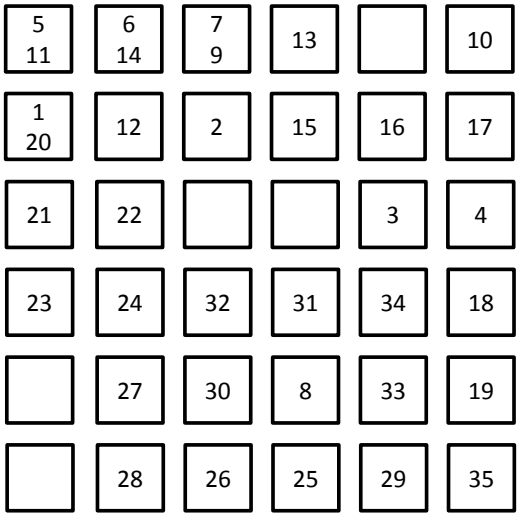

Figure 8: Schedule of physical pages onto 6x6 PLUG

4.5 Extending the Hardware

The extensions to the hardware are relatively straightforward. The scheduling logic in each tile must be extended to allow longer code-blocks, which is essentially the changing of some decoding bits. For complex application like EffiCuts, the lookup rate is no longer one request per cycle. Therefore, we can introduce an interface controller which accepts lookup requests at the rate that the chip can sustain.

5. EVALUATION RESULTS

We obtained an implementation of the EffiCuts algorithm used in [19] from its authors. We modified their implementation as outlined in Section 4.2. We used the PLUG toolchain to implement decision tree based packet classification algorithm on PLUG, as described in Section 3. We modeled our Integer Linear Program for scheduling code-blocks on to PLUG using GAMS [5], a tool for mathematical programming and optimization. The PLUG framework which includes a simulator and power-modeling framework (based on physical design estimates from a prototype implementation in Verilog and synthesis) is used for obtaining performance, area, and power estimates.

We evaluate our design on several metrics including throughput, latency, power consumption and resource utilization. In this evaluation we have used ClassBench [18] for generating synthetic rule sets. We use ClassBench to generate three types of classifiers: Access Control List (ACL), Firewall (FW) and IP Chain (IPC). We also vary the number of rules in a classifier, in particular we generated classifiers with 10,000 and 100,000 rules.

We first present the throughput, latency and power consumption results for our PLUG-based implementation of the packet classification application. Then, we show on-chip resource utilization. Finally, we compare our design against a couple of state-of-art packet classification systems.

5.1 Throughput, Latency and Power

Our implementation has two different blocks: one code-block deals with non-leaf nodes (hence forth referred to as the internal code-block), other one deals with the leaf nodes. We determined the number of instructions that would get executed on the longest

Category	ACL	FW	IPC
Active Physical Pages	19	29	39
Power Consumption (W)	1.5	2.3	3.1

Table 2: Power consumption

code path between the entry and exit points of the code-blocks. We used these numbers to determine the throughput and latency for packet classification achieved by our design.

Internal Code Block. The critical path in the internal code-block consists of about 130 single cycle instructions. It also requires four memory accesses which are scheduled as per the description in section 4.3. This allows a new μcore to start executing the internal code-block (in other words, start processing a new packet) every four cycles.

Leaf Code Block. The longest path in the execution of the leaf code block is dependent on the number of rules associated with leaf. Every two rules require three accesses to the memory. Each rule requires 30 instructions for processing. On the basis of the memory accesses and processing requirement per rule, we concluded that it would be best to have at most four rules per leaf. This means that a leaf code-block requires six memory accesses for accessing four rules. One more access is required for reading the indices of the rules. Hence, in all seven memory accesses are required. By using modulo interleaving to schedule memory accesses, a μcore can start processing a new leaf code-block every seven cycles, i.e. a new packet can be processed every seven cycles.

Throughput. The peak throughput of a PLUG chip operating at a frequency of $1GHz$ is one billion lookup per seconds regardless of the size of PLUG chip or the number of tiles used. To achieve this throughput a new lookup must be initiated every cycle. In our case, the leaf code-block can start a new lookup every seven cycle due to the number of memory accesses required. Hence, we can achieve a throughput of 142 million packets per second. Assuming a minimum packet size of 40 bytes, we can process packets at the rate of 45.4 Gbps. If we consider packet sizes of 64 bytes, throughput is 71 Gbps.

Latency. The latency per packet is dependent on the height of the decision tree (h), the maximum number of rules associated with a leaf (r) and the length of path (l) from the input node to the output node. h decides number of internal code-blocks that executed. r decides the length of the leaf code-block. l decides the time spent in network traversal.

In our design, an internal code-block requires 130 cycles to process which child node to visit next. The leaf code-block needs 30 cycles to perform matching with each rule. Another 30 cycles are required for overhead involved in execution of a leaf code block such as processing input/output messages and loading its data structure. Therefore, the overall latency is given by: $latency = 130h + 30r + 30 + l$. For $h = 6$ and $r = 4$, $l = 16$, we get the latency as $0.95\mu s$. These h, r, and l represent the worst case values among all the classifiers we considered.

Power Consumption. We use PLUG's power model [11] to compute the power consumption (excluding interface power). The model calculates power consumption based on the average number of physical pages required to process a packet. It is based on worst-case power estimates for a tile by considering the synthesized netlist of the PLUG's RTL implementation and the output of Synopsys Design Compiler. The model uses CACTI for calculating power consumed by the memories. Table 2 shows the power consumption for classifiers with 100K rules. We used a 7×7 PLUG chip for FW and IPC and a 6×6 for ACL.

5.2 Resource Utilization

In this section, we explore on-chip resource utilization. We convert EffiCuts trees to PLUG's data flow graph and use the optimizer to schedule them on the PLUG chip. Table 3 shows utilization of on-chip resources (The numbers in parenthesis correspond to actual size). We also report the minimal dimension of PLUG chip needed to accommodate the data flow graph of each classifier. This shows that the scheduling optimizer is able to minimize the dimension of PLUG chip and achieve very high tile utilization in most cases.

From the results of the previous experiment, we also calculate percentage of CPU core, and memory utilization. The percentage is calculated over the actual number of tiles used by the classifiers. Table 3 shows that we are able to achieve good CPU core utilization. However, memory bank utilization numbers are much weaker and since most of the PLUG's area consists of memories (each tile has 256KB of memory), this could be a concern. The very poor memory utilization numbers for the 10K rule cases are due to the small size of the data structures. But even for the 10 times larger 100K cases, utilization is still only between 53% and 64%. The reason is that even when decision trees are large, the first few levels are typically too small to fill a memory bank. Since there are enough small logical pages, the overall memory utilization is low.

This result highlights a generic problem with the PLUG architecture: the fact that creating fine-grained logical pages incurs high memory overhead. We can minimize this problem through changes to the algorithm or to the programming model. The first approach is to change the decision tree algorithm to perform more cuts at the root thus increasing the size of the root and decreasing the size of other levels. The second approach is to share memory banks between multiple logical pages using our modulo interleaving technique to ensure that no conflicts for memory accesses arise. We leave the evaluation of these techniques as future work.

5.3 Comparison Against Other Designs

We now compare the performance of our design to three other state-of-art packet classification systems: Storm [13] which represents a completely software-based approach, Jiang and Prasanna's work [9] which represents an FPGA-based solution, and TCAM's based on Netlogic's state-of-the-art NLA9000 family.

Storm. Ma et al. developed Storm [13], a system for packet classification on software routers. On a 8-core Intel Xeon machine, Storm achieves a throughput of 15 Gbps for 64 byte packets. Assuming 40 byte packets, Storm's throughput is likely to fall below 10 Gbps. This is much below the throughput that our system achieves. Given Storm's performance, we believe that software routers are not suitable for packet classification. Further, Storm is driven by the principle of exploiting locality, which industry has had reluctance to embrace. Power consumption of Storm is also higher than that required by our design. We acknowledge that Storm consists of the full system, while our design only carries out packet classification.

FPGA-based Design. Jiang and Prasanna [9] presented an FPGA-based pipelined architecture for packet classification using decision trees. Their design can sustain 80 Gbps throughput for 40 bytes packets. This throughput was achieved for *ACL* 10K ruleset. On this 10K ruleset, PLUG's performance is about 10% *worse*. Exact power numbers are not available from their system. But since theirs is an FPGA implementation, we expect power consumption to be worse than PLUG's. This is because FPGA's synthesize their func-

Category	ACL		FW		IPC	
Rule Size	10k	100k	10k	100k	10k	100k
Minimal Dimension	5	6	4	7	6	7
Tile Utilization	76% (19)	86% (31)	100% (16)	92% (45)	97% (35)	82% (40)
Core Utilization	66% (400)	56% (560)	72% (368)	56% (800)	81% (912)	63% (816)
Memory Utilization	13% (668 KB)	63% (5,122 KB)	14% (606 KB)	64% (7,537 KB)	7% (611 KB)	53% (5,540 KB)

Table 3: Tile, core and memory utilization

tionality from logic-slice, while PLUG uses specialized but simple processors.

The more interesting case is to look at larger rulesets. We believe that FPGA design is not scalable to 100K rules. This is because FPGAs are highly area inefficient in terms of SRAM storage. A PLUG chip with 64 tiles provides 16MB of storage, thus supports 100K rule sets easily, and occupies a die area of $232mm^2$. Considering the platform they used which is Xilinx Virtex-5 XC5VFX200T. The maximum amount of SRAM storage it provides is 2.5MB and is unlikely to support 100K rules. Since Virtex-5 is one of the largest FPGAs[2], we assume its die size is the maximum possible (25mm x 25mm). Scaling its die size to the $55nm$ technology node gives a die size of $446mm^2$. Such a chip still has *only* 2.5MB SRAM and cannot support 100K rules, even though the die-size is twice of that of a PLUG chip. We acknowledge here that there are various reason for poor SRAM density on FPGAs. And FPGAs are a commodity part compared to PLUGs which are specialized lookup processors.

TCAM. The NLA9000 family of processors from Netlogic [1] represent state-of-the-art TCAM-based solutions for packet classifications. They provide support for 250K rule-sets and can sustain a throughput of 400 million decisions per second. Their power consumption is about 7 Watts and likely to be higher if operated like a "basic" TCAM. In comparison, our PLUG-based system to support similar rule-sets, provides 1/3rd the throughput at 1/5th the power consumption. The more important distinction is that, the PLUG chip's die area is projected to be 131 mm^2. The TCAM die area is hard to exactly quantify - but is likely to be larger, since a TCAM cell is larger than an SRAM cell.

5.4 Limitations and extensions

Our work does not address incremental updates to the packet classification database. We note that the original EffiCuts software does not support incremental updates. If such code for updating the database were available, it could be adapted to work with the PLUG which supports multiple code blocks for each page: not just the code block for lookups, but also others for updates. This allows PLUG update operations to flow through the pipeline without any conflicts with lookups. One update to the database would be broken into multiple smaller PLUG updates submitted to the pipeline. Yet, two challenges remain: coherency between control plane and data plane and the allocation of memory for new nodes. Coherency requires that lookups that are mixed with the many PLUG updates that make up the database update return valid results: the correct answer with the old database or the correct answer with the updated database. Some operations such as updating the interval boundaries for an equi-dense node are too complex to perform in a single PLUG update and leaving the node in an inconsistent state between PLUG updates can lead to incorrect results for the intervening lookups. We can avoid this problem by building a separate inactive copy of the node being updated through as many PLUG updates as needed and atomically flipping the pointer to it in the parent ensuring coherency. The second challenge is incremental memory allocation. As long as there is enough unused memory in the tiles holding the siblings of a new node, creating a new node is not a problem. But if an update requires adding new physical pages, we need to do incremental scheduling to find a suitable place for the new physical page. We leave it to future work to evaluate to what extent this can be done without stopping the lookups while the new database is being placed onto PLUG.

Building on our work with EffiCuts we can forecast with some level of confidence how well other packet classification algorithms would map to PLUG. The challenges imposed by other decision-tree-based algorithms [6,14,13] are similar to those of EffiCuts and we estimate that the porting effort would be similar. Cross-producting-based algorithms [5,16] require hash tables which have been used in the original PLUG applications, so we anticipate that they would not pose significant challenges. The original Lucent bit vector algorithm [11] relies on hardware parallelism to solve the packet classification problem: 5 bitmaps of size equal to the number of rules need to be and-ed together to find the matching rule (when classification is on five fields). PLUG is an architecture with large memory bandwidth and the bitmaps could be distributed among the tiles. Assuming a 6 by 6 PLUG with say 30 of the 36 tiles available for these bit vectors, by reading five 16-bit bitmap chunks from each of the 4 memories in these tiles, we could support databases of up to 30*4*16=1,920 rules at 1GHz/5 = 200 million packets per second. More advanced bit vector algorithms [1] that reduce the memory bandwidth requirement by adding hierarchy would be able to accommodate larger databases. As long as the number of memory reads from each portion of the data structure can be bounded statically, we do not foresee any major difficulties mapping to PLUG. It is hard to estimate the memory utilization efficiency of other algorithms without doing the actual mapping to PLUG.

6. LESSONS LEARNED

Co-design Lessons Designing algorithms without keeping any eye on the hardware on which the algorithm would be executed can result in sub-optimal performance. In our case, we changed the EffiCuts algorithm so that the regions in rule space have dimensions that are a power of 2. This simple change to the algorithm had a significant payoff. The original code-block length was 200 cycles which got reduced to 130 cycles, saving 35% cycles per code-block.

We started off mapping decision trees to PLUG chip. During this process, we realized that PLUG's scheduling constraints are overly conservative and can be relaxed with out losing the guarantees on static scheduling. This is true not only for our application but for all PLUG-based applications. Our integer linear programming formulation for scheduling physical pages to PLUG tiles can be applied

[2]As mentioned in the introduction, this is inherently hard to estimate. But there is widespread consensus that the largest FPGA parts are reticle limited and in this ballpark for die size.

to data flow graph for any application. The algorithm can decide whether a given data flow graph can be mapped to a given PLUG chip.

With minor changes to the hardware and lowering the throughput, we can perform more complex processing. Despite this fact, our design for packet classification still compares favorably against the FPGA-based solution as illustrated in the evaluation section. This suggests, the PLUG architecture has the appropriate set of primitives for lookup processing.

Architecture Lessons Decision trees are highly irregular in nature. This is an artifact of uneven distribution of rules in the rule space. Some trees are very deep, others are very wide. Even within a tree, variations can be seen. But still the PLUG architecture is successfully able to absorb all irregularities. This highlights flexibility of the architecture i.e. PLUG can efficiently support a wide spectrum of applications. The caveat is that each application may require some intelligent design choices before the architecture can be exploited to the fullest.

This also means that PLUG has a balanced architectural design. We analyzed the set of hardware changes required to provide a significant boost to the throughput of the application. Our analysis showed we would need to beef up all the available resources i.e. memory ports, μcores, and on-chip network routers. Thus no single resource is a bottleneck in the design.

Applications Lessons All applications supported by PLUG thus far have been small and hence were amenable to the dataflow programming model. Decision-tree based packet classification is the first major application which resulted in very complex dataflow graphs. But it only took *90 person-hours* to move from nothing to a logical dataflow graph for the application with all the code-blocks programmed. This shows that dataflow programming is as easy (or hard) as any other paradigm.

The memory utilization for the tiles on PLUG chip is not uniform. It is especially low for tiles that map to the upper levels of the decision trees. We outlined specific algorithm design techniques and changes to the programming model that alleviate this problem (Section 5.2), but we leave the evaluation of these techniques as future work.

7. RELATED WORK

Over the years, the problem of packet classification has been studied in great detail. It is still an open problem because solutions need to scale with increasing traffic rates and sizes of rules sets. Power is also increasingly becoming a first class constraint. Many solutions have been proposed in the literature. From hardware perspective, these can be classified based on storage technology into two distinct classes: SRAM-based and TCAM-based.

SRAM-based solutions use different algorithmic approaches for solving the problem. We use the taxonomy used by Song and Turner [16]. As per their classification, algorithms like HiCuts [7], HyperCuts [15], HyperSplits [14] and EffiCuts [19] split the rule set to construct decision trees. Earlier in the paper, we discussed this approach in detail.

Approaches like Bit Vector [12] and Aggregated Bit Vector [2] classify packets by producing bit vectors for each dimension which are intersected together to produce the final result. Combining bit vectors requires storage proportional to the square of the number of rules. Hence, these algorithms do not scale with the size of the rule set. Cross-producting approaches like RFC [6] and DCFL [17] also combine results from different dimensions using cross- product tables. They also suffer from the problem of prohibitive memory required as the rule set size grows.

Present day routers use TCAMs for packet classification for providing very high throughput. But basic TCAMs are very expensive in terms of power, area and dollar cost. Improvements have been proposed in several works [20, 3, 21].

Finally, we discuss some of the approaches taken in recently proposed implementations. Kennedy [10] presented a SRAM-based implementation of HyperCuts on FPGA. Similar to ours, they modified the original algorithm to make it more efficient on hardware. However, they rely on custom logic and achieve low power by reducing clock frequency to match traffic flow. On the other hand, we rely on PLUG platform and are able to achieve low power while retaining high throughput. LOP [8] is an SRAM-based implementation which utilizes an exhaustive search technique similar to TCAM. It utilizes custom logic to compare multiple packets with parts of all rules simultaneously. It consumes lesser power compared to basic TCAM due to reduced number of memory accesses. Similar to our work, it showed that TCAM can be replaced by coupling multiple computation logics with SRAM. However, PLUG is more flexible as it can handle other types of lookup processing as well.

8. CONCLUSION

This paper presented a report on our experiences in designing a new packet classification solution. Its not a *customized* FPGA solution or an implicitly customized ASIC, or an algorithm. Instead we report on a co-design exercise of looking at a lookup processor and algorithms simultaneously to realize the final platform. Our system uses decision-trees generated by the EffiCuts algorithm and maps it to the PLUG architecture. The paper describes novel mechanisms developed at the algorithm, programming model, compiler, and hardware level. Our final results show we can sustain a throughput of 143 million decisions per second with a mere 3 Watts of power.

Our design experience leads to concluding remarks and thoughts along two directions. i) Viability of PLUG as a lookup processor. ii) Co-designing and the necessity for a platform-level view of algorithms through hardware design details for packet classification for lookup processors.

When we started the PLUG project we used some intuitive and reasonably complex examples to flesh out the design and drive our search for what primitives should go into the architecture of a lookup processor. We intentionally avoided overly complex examples so that we could make progress on several fronts simultaneously. Previous papers described the architecture and showed it can perform well on *simple* matching problems [4, 11]. In this paper, we report on our experience in mapping perhaps the most challenging matching problem - packet classification. It was initially demoralizing to learn the *original* PLUG architecture simply could not support EffiCuts. Simple modifications to its execution rules, some novel improvements to its programming model and compiler toolchain, and simple changes to the hardware allowed the mapping of packet classification. In the end, the impressive combination of performance and power results, was surprising to us. We now confidently believe, the PLUG system is indeed balanced and has the right set of primitives. A commercial deployment of such a system will undoubtedly have to tweak some parameters, but the overall architecture appears sound and truly capable of handling complex workloads.

For the architecture and its programming model, the fact that we were able to implement and map the newest and *best* packet classification algorithm on the architecture shows it does live up to

its promise of flexibility. The fact that, we are able to beat perhaps the best customized algorithmic packet classification solution in the literature [9], shows the entire platform is efficient.

When we started the study our goal was to also understand if algorithmic efforts that abstracted away details of the hardware or implicitly assumed an ASIC, would require significant rethinking for implementing on a lookup processor. Looking at the entire platform meant several person-hours. We had to look at implementation at a level of detail and breadth in terms of algorithm, compiler, and architecture. While time consuming, it allowed us to quantitatively understand the implications of algorithms. We feel most of the algorithmic work, while abstracting away hardware details, implicitly makes the right set of assumptions. Algorithmic work can abstract away all computation and SRAM as a black-box. Understanding the implications of SRAM line-width, size of accesses, number of accesses, complexity of the computation and its potential impact on hardware area and complexity can help simplify the process of migrating the algorithmic work to eventual platforms. In the end, we feel lookup processor can flexibly provide the primitives that these algorithms desire.

9. ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd Michela Becchi for comments and feedback. Thanks to Balajee Vamanan, Gwendolyn Voskuilen, and T. N. Vijaykumar for providing us access to the Efficuts source code and explanations. Support for this research was provided by the National Science Foundation under the following grants: CCF-0845751, CCF-0917238, CNS-0917213 and the Wisconsin Alumni Research Foundation.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

10. REFERENCES

[1] Layers 2-4 knowledge-based processors, http://www.netlogicmicro.com/products/layer4/index.asp.

[2] F. Baboescu and G. Varghese. Scalable Packet Classification. *IEEE/ACM Transactions on Networking*, Feb. 2005.

[3] E. S. David, D. Taylor, and J. Turner. Packet Classification Using Extended TCAMs. In *Proceedings of International Conference on Network Protocols (ICNP)*, 2000.

[4] L. De Carli, Y. Pan, A. Kumar, C. Estan, and K. Sankaralingam. PLUG: Flexible Lookup Modules for Rapid Deployment of New Protocols in High-speed Routers. SIGCOMM '09.

[5] GAMS. `http://www.gams.com`.

[6] P. Gupta and N. McKeown. Packet Classification on Multiple Fields. In *SIGCOMM*, 1999.

[7] P. Gupta and N. McKeown. Classifying Packets with Hierarchical Intelligent Cuttings. *IEEE Micro*, January 2000.

[8] X. He, J. Peddersen, and S. Parameswaran. LOP: A Novel SRAM-based Architecture for Low Power and High Throughput Packet Classification. CODES+ISSS '09.

[9] W. Jiang and V. K. Prasanna. Large-scale Wire-speed Packet Classification on FPGAs. FPGA '09.

[10] A. Kennedy, X. Wang, Z. Liu, and B. Liu. Low Power Architecture for High Speed Packet Classification. ANCS '08.

[11] A. Kumar, L. De Carli, S. J. Kim, M. de Kruijf, K. Sankaralingam, C. Estan, and S. Jha. Design and Implementation of the PLUG Architecture for Programmable and Efficient Network Lookups. PACT '10.

[12] T. V. Lakshman and D. Stiliadis. High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching. SIGCOMM '98.

[13] Y. Ma, S. Banerjee, S. Lu, and C. Estan. Leveraging Parallelism for Multi-dimensional Packet Classification on Software Routers. SIGMETRICS '10.

[14] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li. Packet Classification Algorithms: From Theory to Practice. In *INFOCOM 2009*.

[15] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet Classification using Multidimensional Cutting. SIGCOMM '03.

[16] H. Song and J. Turner. Toward Advocacy-Free Evaluation of Packet Classification Algorithms. *IEEE Trans. Comput.*, May 2011.

[17] D. Taylor and J. Turner. Scalable Packet Classification using Distributed Crossproducting of Field Labels. In *INFOCOM 2005*.

[18] D. E. Taylor and J. S. Turner. ClassBench: A Packet Classification Benchmark. *IEEE/ACM Trans. Netw.*, 15, June 2007.

[19] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar. EffiCuts: Optimizing Packet Classification for Memory and Throughput. SIGCOMM, 2010.

[20] F. Zane, G. Narlikar, and A. Basu. CoolCAMs: Power-Efficient TCAMs for Forwarding Engines. In *IEEE INFOCOM*, 2003.

[21] K. Zheng, H. Che, Z. Wang, and B. Liu. TCAM-based Distributed Parallel Packet Classification Algorithm with Range Matching Solution. In *IEEE INFOCOM*, 2005.