LEAP: Latency- Energy- and Area-optimized Lookup Pipeline

Master's Degree Project Report

Eric N. Harris
University of Wisconsin-Madison
920-216-2560 enharris@uwalumni.com

ABSTRACT

Table lookups and other types of packet processing require so much memory bandwidth that the networking industry has long been a major consumer of specialized memories like TCAMs. Extensive research in algorithms for longest prefix matching and packet classification has laid the foundation for lookup engines relying on area- and power-efficient random access memories. Motivated by costs and semiconductor technology trends, designs from industry and academia implement multi-algorithm lookup pipelines by synthesizing multiple functions into hardware, or by adding programmability. In existing proposals, programmability comes with significant overhead.

This report details LEAP, a latency- energy- and area-optimized lookup pipeline based on an analysis of various lookup algorithms. This report describes the architecture and microarchitecture. It compares LEAP to PLUG, which relies on von-Neumann-style programmable processing. It shows LEAP has equivalent flexibility as PLUG while reducing chip area by $1.5\times$, power consumption by $1.3\times$, and latency typically by $5\times$. Furthermore, this report presents an intuitive Python-based API for programming LEAP. This report details completed work and suggests a path for future work on LEAP.

1. INTRODUCTION

Lookups are a central part of the packet processing performed by network switches and routers. Examples include forwarding table lookups to determine the next hop destination for the packet, and packet classification lookups to determine how the given packet is to be treated for service quality, encryption, tunneling, etc. Methods to achieve these lookup operations include: software based table lookups [17], lookup hardware integrated into a packet processing chip [18], and dedicated lookup chips [3, 4]. The latter two are the preferred industry approach.

Trends indicate an increasing sophistication in the lookup processing required and reducing benefit from technology scaling. Borkar and Chien show that energy efficiency scaling of transistors is likely to slow down, necessitating higher-level design innovations that provide energy savings [9].

This master's project is the culmination of a year's worth of collaborative work on a new class of flexible lookup engines with reduced latency, energy consumption, and silicon area. These savings ultimately translate into cost reductions or more aggressive scaling for network equipment as described below.

1. Latency: Lookup engine latency affects other components on the router interface. The exact nature of the savings depends on the line card architecture, but it can result in a reduction in the size of high-speed buffers, internal queues in the network processor, and the number of threads required to achieve line-speed operation. A major reduction in the latency of the lookup engine can indirectly result in important area and power savings in other chips on the line card.

2. Energy/Power: Reducing the power consumption of routers and switches is in itself important because the cost of electricity is a significant fraction of the cost of operating network infrastructure. Even more important, reducing power improves scalability because the heat dissipation of chips, and the resulting cooling challenges, are among the main factors limiting the port density of network equipment. Our design, LEAP, demonstrates energy savings for lookups through architectural innovation.

3. Area: Cutting-edge network processors and stand-alone lookup engines are chips of hundreds of square millimeters. Reducing the silicon area of these large chips results in a super-linear savings in costs.

The first part of the project involved an initial investigation and characterization of lookup processing (see Section 2). After this investigation, an architecture called LEAP was proposed. It is a latency- energy- and area-optimized lookup pipeline architecture that retains the flexibility and performance of earlier proposals for lookup pipeline architectures while significantly reducing the overheads that earlier proposals incur to achieve flexibility. Instead of programmable microengines, LEAP uses a dynamically configurable data path. The initial investigation included analysis of seven algorithms for forwarding lookups and packet classification to determine a mix of functional units suitable for performing the required processing steps. LEAP's high

level architecture is described in Section 4. The most time consuming part of the LEAP project was an actual implementation of the LEAP design. The microarchitecture of this implementation is described in Section 5. This specific design was implemented in RTL, verified and synthesized it to a 55nm ASIC library. PLUG[13] is an earlier proposal for a tiled smart memory architecture that can perform pipelined lookups. At the same technology node, LEAP achieves the same throughput as PLUG and supports the same lookup algorithms but has $1.5\times$ lower silicon area, and $1.3\times$ lower energy. Latency savings depend on the lookup algorithms used: we observe between $1.7\times$ and $6.5\times$, with typical savings exceeding $5\times$.

This project's contributions are:

- Characterization of lookup processing (Section 3)
- LEAP architecture (Section 4)
- A working LEAP implementation and microarchitecture (Section 5)
- LEAP Verification Framework (Section 6)
- Discussion of tradeoffs made in LEAP (Section 8)
- Discussion of physical design considerations (Section 7)
- Programming mechanisms and a Python API (Section 9)
- Quantitative evaluation of LEAP (Section 10)
- A pending patent and a framework for future work (Section 11)

2. BACKGROUND AND MOTIVATION

To put the LEAP project in context, this section describes existing works. At the top-level there are run-to-completion (RTC) architectures and dedicated lookup-based engines. In the RTC paradigm, lookup data is maintained by the network processor on a globally shared memory (across the processor's cores), and it exploits packet-level parallelism. The core in charge of each packet performs the lookup based on the data in the packet by looking up a globally shared datastructure. Examples include Cisco's Silicon Packet Processor [18]. An alternative is dedicated lookup engines interfaced with the network processor. These engines are organized as a pipeline where, at each pipeline step (which is a hardware block), packet bits are moved close to relevant lookup data. The goal is to minimize the amount of data moved around. In principle, both approaches are equally valid and in this work focuses on systems that belong to the latter class of dedicated lookup-based engines.

Within this domain, two main approaches exist for implementing lookups: i) relying on massive bit-level hardware parallelism and ii) using algorithms.

2.1 Bit-level hardware parallelism

In this approach, typically a ternary content-addressable memory (TCAM) is employed to compare a search key consisting of packet header fields against all entries of the table in parallel. Due to low density of TCAM storage and power challenges, much research effort has focused on the second approach of finding good RAM-based algorithmic solutions.

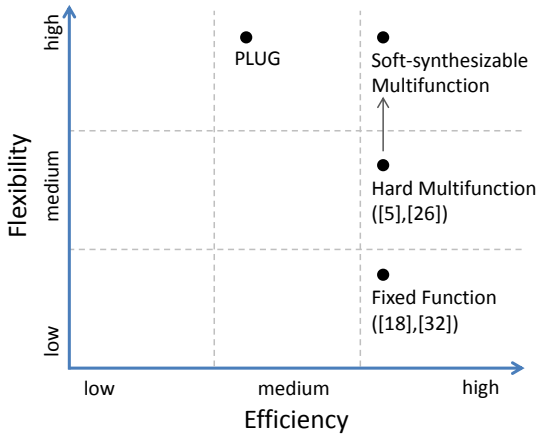


Figure 1: Design space of Lookup Engines

2.2 Algorithmic Lookup Engine Architectures

The main challenge for lookup algorithms is minimizing the amount of memory required to represent the lookup table while keeping the number of memory references low and the processing steps simple as described in a mature body of work [22, 32, 38, 6]. The common characteristics across different types of lookups are the following: performance is dominated by frequent memory accesses with poor locality, processing is simple and regular, and plentiful parallelism exists because many packets can be processed independently. Here each lookup is partitioned into individual tasks organized in a pipeline, and multiple packets can be concurrently processed by a single lookup engine by pipelining these tasks across the packets. Figure 1 plots lookup engines classified according to flexibility and efficiency.

Fixed-Function: Specialized approaches where a chip is designed for each type of lookup provide the most efficiency and least flexibility. In the FIPL architecture [33], an array of small automata are connected to a single memory to provide an efficient specialized architecture for the IP forwarding lookup problem. By placing the processing close to memory and using specialized hardware, high efficiency is achieved. Such specialized algorithmic lookup engines are widely used both as modules inside network processors, for example, in QuantumFlow [12] and as stand-alone chips [3, 4] acting as a coprocessors to network processors or other packet processing ASICs. The fixed-function approach is often implemented with partitioned memories accompanied by their own dedicated hardware modules to provide a pipelined high-bandwidth lookup engine [27].

Hard Multi-Function: Versatility can be increased without compromising efficiency by integrating different types of fixed-function processing that all use the same data-paths for memories and communication between pipeline stages. Baboescu *et al.*[5] propose a circular pipeline that supports IPv4 lookup, VPN forwarding and packet classification. Huawei is building tiled Smart Memories [29] that support 16 functions including IP forwarding lookup, Bloom filters that can be used in lookups [16, 15], sets and non-lookup functions

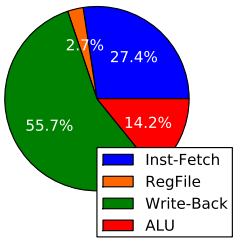


Figure 2: Programmable processor energy breakdown.

such as queues, locks, and heaps. As is common for industry projects, details of the internals of the architecture and programming model are not disclosed.

Specialized Programmable: Hard multi-function lookup pipelines lack the flexibility to map lookups whose functionality has not been physically realized at manufacture time. The PLUG [13] lookup pipeline achieves flexibility by employing principles from programmable von-Neumann style processors in a tiled architecture. Specifically, it simplifies and specializes the general purpose processor and processing is achieved through 32 16-bit microcores in each tile that execute processing steps of lookup algorithms based on instructions stored locally. Communication is provided by six separate 64-bit networks with a router in each tile. These networks enable non-linear patterns of data flow such as the parallel processing of multiple independent decision trees required by recent packet classification algorithms such as Efficuts [38].

While PLUG attains high flexibility, it lacks high efficiency. It is instructive to understand why it has inefficiencies. Figure 2 shows the percentage contribution of energy consumption of a 32-bit in-order core similar to PLUG as estimated with the McPAT [31] modeling tool. Events other than execute are *overhead* yet they account for more than 80% of the energy. While it may seem surprising, considering that a single pipeline register consumes 50% of the energy of an ALU operation and staging uses three pipeline registers for PLUG's 3-cycle load, these overheads soon begin to dominate. Others have also made this observation. In the realm of general purpose processors, Hameed *et al.*[24] recently showed how the von-Neumann paradigm of fetching and executing at instruction granularity introduces overheads.

2.3 Obtaining Efficiency and Flexibility

With over 80% of energy devoted to overhead, it appears the von-Neumann approach has far too much inefficiency and is a poor starting point. The specific inefficiencies in von-Neumann lookup engine designs were investigated as part of this project. This investigation (detailed in Appendix A) showed that a von-Neumann approach had at least 100x the computational area of an ASIC performing the same lookup engine functions. In terms of dynamic power, ASICs use as low as 40% the power of a von-Neumann lookup engine. The existing von-Neumann approaches also add latency for a number of reasons including the wasted time spent storing values to and from the register file. These inefficiencies motivated this project's search for a more efficient flexible lookup engine.

Due to fundamental energy limits, recent work in general purpose processors has turned to other paradigms for improving efficiency. This shift is also related to our goal and can be leveraged to build efficient and flexible *soft-synthesizable multi-function* lookup engines. LEAP draws inspiration from recent work in hardware specialization and hardware accelerators for general purpose processors that organizes coarse-grained functional units in some kind of interconnect and dynamically synthesizes new functionality at run-time by configuring this interconnect. Some examples of this approach include: DySER [21], FlexCore [36], QsCores [39], and BERET [23]. These approaches by themselves cannot serve as lookup engines because they have far too high latencies, implicitly or explicitly rely on a main-processor for accessing memories, or include relatively heavyweight flow-control mechanisms and predication mechanisms internally. Instead, LEAP leverages their insight of dynamically synthesizing functionality, combine it with the unique properties of lookup engine processing and develop a stand-alone architecture suited for lookup engines.

FPGAs provide an interesting platform and their inherent structure is suited for flexibility with rapid and easy run-time modification. However, since they perform reconfiguration using fine-grained structures like 4- or 8-entry lookup tables, they suffer from energy and area efficiency and low clock frequency problems. Also, a typical FPGA chip has limited amounts of SRAM storage. While they may be well suited in some cases, they are inadequate for large lookups in high-speed network infrastructure.

3. TOWARD DYNAMIC MULTI-FUNCTION LOOKUP ENGINES

This section characterizes the computational requirements of lookups. Assuming a dataflow approach where processing is broken up in steps and mapped to a tile based pipeline similar to PLUG, this section focuses on the processing to understand how to improve on existing flexible lookup approaches and incorporate the insights of hardware specialization. Much of this work was based off a coursework project (see Appendix A).

3.1 Description of Lookup Algorithms

The LEAP architecture was designed to be generic enough to support common types of lookup operations. It was developed by examining seven representative lookup algorithms in detail. Each algorithm is used within a network protocol or a common network operation. Table 1 summarizes the application where each lookup algorithm originated, and the key data structures being used. Most analysis was in understanding the basic steps of the lookup processing and determining the hardware requirements by developing de-

Context	Approach/Key data structures
Ethernet forwarding	D-left hash table [8, 10, 40]
IPv4 forwarding	Compressed multi-bit tries [14]
IPv6 forwarding	Compressed multi-bit tries + hash tables [26]
Packet classification	Efficuts with parallel decision trees [38, 37]
DFA lookup	DFA lookup in compressed transition tables[28]
Ethane	Parallel lookup in two distinct hash tables [11]
SEATTLE	Hash table for cached destinations, B-tree for DHT lookup [25]

Table 1: Characterization of lookup algorithms.

sign sketches as shown in Figure 3. This report describes this analysis in detail for two algorithms, Ethernet forwarding and IPv4 and the next section summarizes overall findings. These two serve as running examples through this paper, with their detailed architecture discussed in Section 4.3 and programming implementation discussed in Section 9.

Ethernet forwarding

Application overview: Ethernet forwarding is typically performed by layer-II devices, and requires retrieving the correct output ports for each incoming Ethernet frame. This implementation uses a lookup in a hash table that stores, for every known layer-II address (MAC), the corresponding port.

Ethernet lookup step: This step, depicted in Figure 3a, checks whether the content of a bucket matches a value (MAC address) provided externally. If yes, the value (port) associated with the key is returned. To perform this, a 16-bit *bucket id* from the input message is used as a memory address to retrieve the bucket content. The bucket key is then compared with the key being looked up, also carried by the input message. The bucket value is copied to the output message; the message is sent only if the the two keys match.

IPv4 lookup

Application overview:

The IPv4 forwarding approach discussed here is derived from PLUG, and is originally based on the "Lulea" algorithm [14]. This algorithm uses compressed multibit tries.

In multibit tries, each node represents a fixed number of prefixes corresponding to each possible combination of a subset of the address bits. For example, a stage that consumes 8 bits covers 256 prefixes. In principle, each prefix is associated with a pointer to a forwarding rule and a pointer to the next trie level. In practice, the algorithm uses various kinds of compressed representations, avoiding repetitions when multiple prefixes are associated with the same pointers. The following describes one of the processing steps used in IPv4.

IPv4 rule lookup step: This step, represented in Figure **??**b, uses a subset of the IPv4 address bits to construct a pointer into a forwarding rule table. Specifically, it deals with the case where the 256 prefixes in a trie node are partitioned in up to seven ranges, each associated with a different rule. In the rule table, the seven rules are stored sequentially starting at a known base offset. The goal of this step is to select a range based on the IPv4 address being looked up and construct a pointer to the corresponding rule.

Initially, 16 bits from the input message (the *node id*) are used as an address to retrieve a trie node from memory. The node stores both the base offset for the rules, and the seven ranges in which the node is partitioned. The ranges are represented as a 3-level binary search tree. Conceptually, the lookup works by using 8 bits from the IPv4 address as a key, and searching the range vector for the largest element which does not exceed the key. The index of the element is then added to the base offset to obtain an index in the rule table. Finally, the result is forwarded to the next stage.

3.2 Workload Analysis

Similar to the above two, students working on LEAP analyzed many processing steps of several lookup algorithms. This analysis revealed common properties which present opportunities for dynamic specialization and for eliminating von-Neumann-style processing overheads.

1. Compound specialized operations: The algorithms perform many specific bit-manipulations on data read from the memory-storage. Examples include bit-selection, counting the bits set, and binary-space partitioned search on long bit-vector data. Much of this is efficiently supported with specialized hardware blocks rather than through primitive instructions like `add`, `compare`, `or`, etc. For example, a single `bstsearch` instruction that does a binary search through 16-bit chunks of a 128-bit value, like used in [30], can replace a sequence of `cmp, shift` instructions. There is a great deal of potential for reducing latency and energy with simple specialization.

2. Significant instruction-level parallelism: The lookups show opportunity for instruction-level parallelism (ILP), i.e. several primitive operations could happen in parallel to reduce lookup latency. Architectures like PLUG which use single-issue in-order processors cannot exploit this.

3. Wide datapaths and narrow datapaths: The algorithms perform operations on wide data including 64-bit and 128-bit quantities, which become inefficient to support with wide register files. They also produce results that are sometimes very narrow: only 1-bit wide (bit-select) or 4-bits wide (bit count on a 16-bit word) for example. A register file or machine-word size with a fixed width is over-designed and inefficient. Instead, a targeted design can provide generality and reduced area compared to using a register file.

4. Single use of compound specialized operations: Each type of compound operation is performed only once (or very few times) per processing step, with the result of one operation being used by a *different compound operation*. A register-file to hold temporary data is not required.

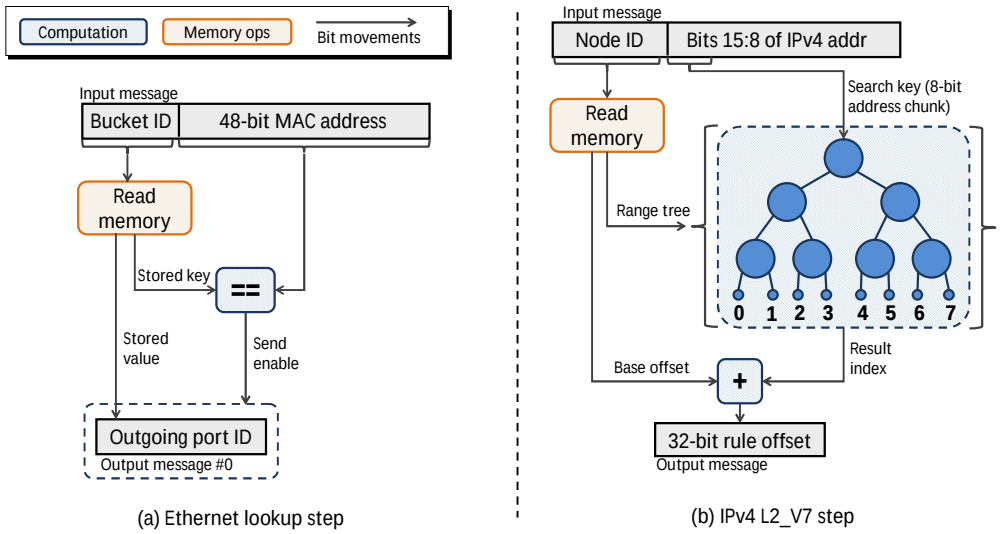


Figure 3: Rule offset computation in IPv4

5. Many bit movements and bit extractions: Much of the "processing" is simply extracting bits and moving bits from one location to another. Using a programmable processor and instructions to do such bit movement among register file entries is wasteful in many ways. Instead, bit extraction and movement could be a hardware primitive.

6. Short computations: In general, the number of operations performed in a tile is quite small - one to four. De Carli et al[13] also observe this, and specifically design PLUG to support "code-blocks" no more than 32 instructions long.

These insights led to a design that eliminates many of the overhead structures and mechanisms like instruction fetch, decode, register-file, etc. Instead, a lookup architecture can be realized by assembling a collection of heterogeneous functional units of variable width. These units communicate in arbitrary dynamically decided ways. Such a design transforms lookup processing in two ways. Compared to the fixed-function approach, it allows dynamic and lookup-based changes. Compared to the programmable processor approach, this design transforms long multi-cycle programs to a single-cycle processing step. The next section describes the LEAP architecture, which is an implementable realization of this abstract design.

4. LEAP ARCHITECTURE

This section describes the high-level architecture and idea of the LEAP lookup engine. First, its organization and execution model are described and its detailed design is discussed. Next, we walk through an example of how lookup steps map to LEAP. A discussion of physical implementation and design tradeoffs concludes this section. The general LEAP architecture is flexible enough to be used to build substrates interconnected through various topologies like rings, buses, meshes etc. and integrated with various memory technologies. This report assumes a mesh-based chip organization and integration with SRAM. Section 5 details the microarchitecture of a working implementation of the high-level LEAP design presented in this section.

4.1 Hardware Organization

For clarity this section assumes all computation steps are one cycle. Section 4.4 relaxes this assumption.

Organization: Figure 4 presents the LEAP architecture spanning the coarse-grained chip-level organization showing 16 tiles and the detailed design of each tile. LEAP reuses the same tiled design as PLUG ([26]) in which lookups occur in steps with each step mapped to a tile.

Each tile consists of a LEAP compute engine, a router-cluster, and an SRAM. At the chip-level, the router cluster in each tile is used to a form a mesh network across the tiles. LEAP mirrors the design of PLUG in which tiles communicate only to their immediate neighbors and the inter-tile network is scheduled at compile time to be conflict-free.

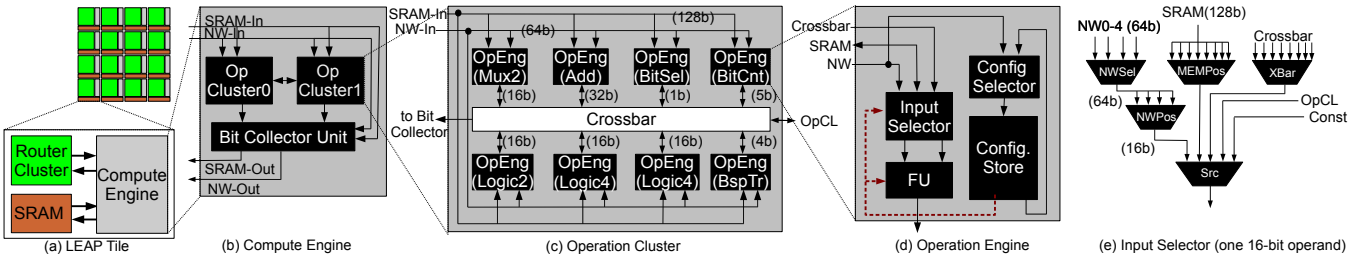


Figure 4: LEAP Organization and Detailed Architecture

Functional Unit	Description
Add	Adds or subtracts two 32-bit values. It can also decrement the final answer by one.
BitCnt	Bitcounts of all or a subset of a 32-bit input
BitSel	Can shift logically or arithmetically and select a subset of bits
BSPTr	Performs a binary space tree search comparing an input to input node values.
Logic2	Logic function "a op b" where op can be AND,OR,XOR,LT,GT,EQ,etc
Logic4	Operates as "(a op b) op (c op d)" or chooses based on an operation: "(a op b) ? c : d"
Mux2	Chooses between 2 inputs based on a input

Table 2: Functional Units Mix in the operation engines

Bits	Description
80	Valid Bit
79:72	Destination Tile ID
71	Config Mode (1 indicates this message should write to the configuration of the tile)
70:64	Codeblock (The type of message)
63:0	Message Data

Table 3: LEAP Network Message Format

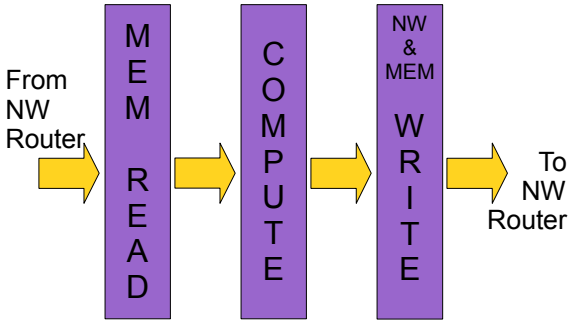


Figure 5: LEAP Pipeline

PLUG used six inter-tile networks, but our analysis showed only four were needed, each network being 64 bits wide. Lookup requests arrive at the top-left tile on the west interface, and the result is delivered on the east output interface of the bottom-right tile. With four networks, LEAP can process lookups that are up to 384 bits wide. Each SRAM is 256KB and up to 128 bits can be read or written per access.

Each LEAP compute engine is connected to the router cluster and the SRAM. It can read and write from any of the four networks, and it can read and write up to 128 bits from and to the SRAM per cycle. Each compute engine can perform computation operations of various types on the data consumed. Specifically, the following seven types of primitive hardware operations are allowed as decided by workload analysis: *select*, *add*, *bitselect*, *bitcount*, *2-input logical-op*, *4-input logical-op*, *bsptree-search* (details in Table 2 and Section 4.2). For physical design reasons, a compute engine is partitioned into two identical operation clusters as shown in Figure 4c. Each of these communicate with a bit-collection unit which combines various bits and sends the final output message. Each operation cluster provides the aforementioned hardware operations, each encapsulated inside an operation engine. The operation engine consists of the functional unit, an input selector, and a configuration-store.

Execution model: The arrival of a *message* from the router-cluster triggers the processing for a lookup request. Based on the *type* of message, different processing must be done. The processing can consist of an arbitrary number of primitive hardware operations performed serially or concurrently such that they finish in a single cycle (checked and enforced by the compiler). The LEAP computation engine performs the required computation and produces results. These results can be written to the memory, or they can result in output messages.

Each LEAP tile is programmed to handle the *types* of messages different, with each *type* having a corresponding *code-block* of processing that is required. The two terms are used interchangably in this report. The message format between tiles can be seen in Table 3.

In this design, a single Compute Engine is sufficient. With this execution model, the architecture sustains a throughput of one lookup every cycle. The architecture was designed to run at 1GHz and thus provide a throughput of 1 billion lookups per second. See Section 10 for how an actual implementation matches up with this architectural design point.

Pipeline: The high-level pipeline abstraction that the organization, execution-model, and compilation provides is a simple pipeline with three stages (cycles): *memory-read (R)*, *compute (C)*, and *memory-write/network-write (Wr)*. For a visual representation, see Figure 5.

For almost all of our lookups, a simple 3-stage (3-cycle) pipeline of *R, C, Wr* is sufficient in every tile. This provides massive reductions in latency compared to the PLUG

approach. In the case of updates or modifications to the lookup table, the *R* stage does not do anything meaningful. LEAP supports coherent modifications of streaming reads and writes without requiring any global locks by inserting "write bubbles" into the lookup requests[7]. The R stage forms SRAM addresses from the network message or through simple computation done in the computation engine. Our analysis showed this sufficient.

Compilation: Lookup processing steps can be specified in a high-level language to program the LEAP architecture. Specifically, there is a Python API developed for LEAP (details in Section 9). As far as the programmer is concerned, LEAP is abstracted as a sequential machine that executes one hardware operation at a time. This abstraction is easy for programmers. The compiler takes this programmer-friendly abstraction and maps the computation to the hardware to realize the single-cycle compute-steps. The compiler uses its awareness of the hardware's massive concurrency to keep the number of compute steps low. The compiler's role is threefold: i) data-dependence analysis between the hardware operations, ii) hardware mapping to the hardware functional units, and iii) generation of low-level configuration signals to orchestrate the required datapath patterns to accomplish the processing. The end result of compilation is simple: a set of configuration bits for each operation engine in each operation cluster and configuration of the bit-collection unit to determine which bits from which unit are used to form the output message, address, and data for the SRAM. This compilation is a hybrid between programmable processors that work on serial ISAs and hardware synthesis.

Note that the compiler for LEAP is not yet finished. Its current status is described in Section 9.

4.2 Design

This section describes the hierarchy and design of the LEAP architecture. For details on implementation, see our working prototype microarchitecture in Section 5. At the top level LEAP is organized into tiles, these tiles each have a compute engine, each compute engine has two operation clusters filled with operation engines. The details are described below:

Tile (Figure 4(a)): A single tile consists of one LEAP compute-engine, interfaced to the router-cluster and SRAM.

Compute engine (Figure 4(b)): The compute engine must be able to execute a large number of primitive hardware operations concurrently while allowing the results from any hardware unit to be seen by any other hardware unit. To avoid introducing excessive delay in the forwarding path, it must perform these tasks at low latency. The workload characterization revealed that different lookups require different types of hardware operations, and they have large amounts of concurrency ranging up to four logical operations in IPv6 for example. Naively placing four copies of each of the eight hardware operations on a 32-wide crossbar would present many physical design problems. To overcome these, LEAP has a clustered design with two identical *operation clusters*, allowing one value to be communicated between the clusters (to limit the wires and delay).

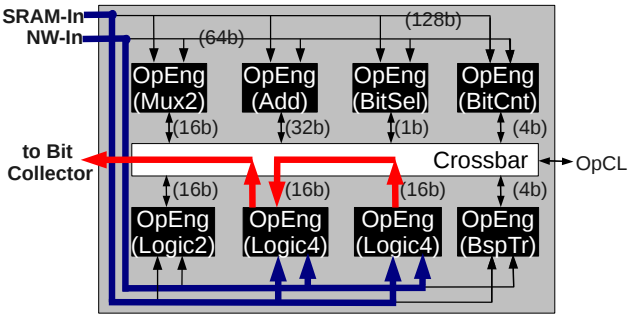
Different lookups combine bits from various operations to create the final output message or the value for the SRAM. To provide this functionality in as general a fashion as possible, the compute engines are interfaced to a bit collector, which receives the operation engine results being fed to it. This unit includes a bit-shifter for the input coming from each operation engine, one level of basic muxing and a 4-level OR-tree that combines all of the bits to produce 64-bit messages, 128-bit value, and 32-bit address for outgoing network messages and SRAM value/address respectively.

Operation cluster(Figure 4(c)): The operation cluster combines eight operation engines communicating with each other through a crossbar. It also receives inputs from and outputs to all four networks and the SRAM. It receives one input from the neighbor operation cluster and produces outputs to the bit collector. Depending on compiler analysis, the crossbar is configured into different datapaths as shown by the two examples in Figure 6. Based on our workload analysis, we found the *4-input logical-op* unit was used the most, hence we provide two of them in each cluster.

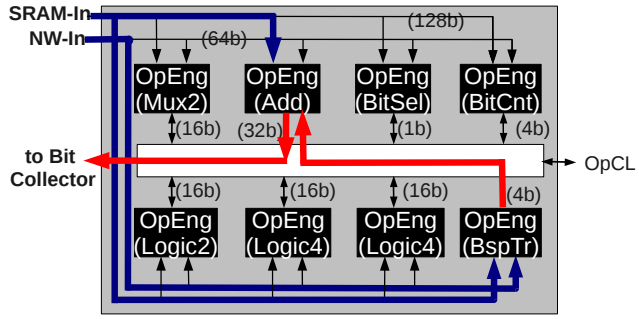
Operation engine(Figure 4(d)): The core computation happens in each operation engine, which includes a configuration store, an input selector, and the actual hardware functional unit like an adder or a comparator. LEAP provides seven types of hardware functional units as described in Table 2. The main insight behind the operation engine is a throwback to micro-controlled machines which encode the control signals into a micro-control store and sequence operations. LEAP effectively has loosely distributed concurrent micro-controlled execution across all the operation engines. Each operation engine must first select its inputs from one of the four networks, values from the SRAM, values from any other operation engine (i.e. the crossbar), or values from a neighboring operation cluster. This is shown by the selection tree in Figure 4(e). Furthermore, the actual inputs delivered to the functional unit can be a subset of bits, sign- or zero-extended, or a constant provided by the configuration store. A final selection step decides this and provides the proper input to the functional unit. The result of the functional unit is sent to the crossbar. The configuration store includes the control signals for all elements in the operation engine. Each operation engine has a different sized configuration vector, depending on the number and type of operands.

Section 9 provides a detailed example showing the Python-API and its compiler-generation configuration information.

Reading the configuration-store to control the operation-engine proceeds as follows. Every cycle, if a message arrives its bits are used to index into the configuration store and decide the configuration to load the controls signals for the



(a) Example Datapath for Ethernet Forwarding mapping a 48 bit equality to 2 Logic4's



(b) Example Datapath for IPv6

Figure 6: Dynamically created datapaths.

operation engine. An important optimization and insight is the use of such pre-decoded control information as opposed to instruction fetch/decode like in von-Neumann processing. By using configuration information, we eliminate all decoding overhead. More importantly, if successive messages require the same compute step, no reconfiguration is performed and no additional dynamic energy is consumed. Further application analysis is required to quantify these benefits, and our quantitative estimates do not account for this.

4.3 Mapping lookups to LEAP's architecture

To demonstrate how lookup steps map to LEAP, we revisit the examples introduced in Section 3.1. Figure 6 shows how the steps shown in Figure 3 are configured to run on LEAP.

In the example Ethernet forwarding step, the *R-stage* reads a bucket containing a key (MAC) and a value (port). The *C-stage* determines if the 48-bit key matches the key contained in the input message. If it matches, the bit collector sends the value out on the tile network during the *memory-write/network-write (Wr) stage*. In order to do a 48-bit comparison, two Logic4 blocks are needed. The first Logic4 can take four 16 bit operands and is fed the first 32 bits (2 operands of 16 bits) of the key from SRAM and the first 32 bits of the key from the input message. This Logic4 outputs the logical AND of two 16-bit equality comparisons. The second Logic4 ANDs the output of the first Logic4 with the equality comparison of the remaining pair of 16 bits to check. The result is sent to the bit collector, which uses the result to conditionally send. Since data flows freely between the functional units, computation completes in one cycle (as

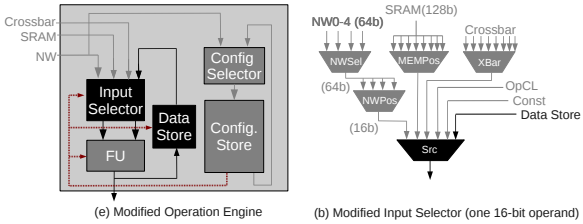


Figure 7: Support for multicycle operation. Added hardware is shown in black.

it also does in the shown IPv6 example). If we assume SRAM latency is 1 cycle and it takes 1 cycle to send the message, LEAP completes both the Ethernet forwarding step and IPv6 step in Figure 6 in 3 cycles. The equivalent computation and message formation on PLUG's von-Neumann architecture would take 10 cycles for the Ethernet forwarding step and 17 cycles for the IPv4 step. With LEAP, computation no longer dominates total lookup delay. These examples are just one step; to complete the lookup the remaining steps are mapped to other tiles in the same manner.

4.4 Multi-cycle compute step

The LEAP architecture can easily be extended to handle sophisticated lookup operations. For example, the packet classification application requires 8 memory reads and more than 30 compute steps.

Conceptually, the LEAP hardware breaks the processing into *multi-cycle compute steps*. The pipeline is extended to contain multiple *C* stages. Each compute step executes in a single cycle. These details are abstracted from the programmer and are handled by the compiler which generates low-level information saved into the config-store. The design presented earlier is augmented in a few simple ways as outlined in Figure 7. All changes are restricted to the operation engine.

First, to ensure that reads and updates are still coherent and correctly serialized, the compiler lengthens all compute steps in a tile into *multi-cycle compute steps* and delaying the *W* stage to match the longest one.

Second, LEAP needs a mechanism to carry values between compute steps. Each operation engine is augmented with a data store as shown in Figure 7. From our empirical analysis, a 4-operand store was sufficient and has similarities to a register file in general purpose processors and PLUG micro-cores. However, it is physically associated with a single functional unit and hence avoids energy overheads in communication. It is also much smaller in size. In spirit, it is similar to the register file cache that Gebhart et al. recently proposed for GPUs to reduce register file access energy [20, 19]. To support reads from this data store, the configuration store of each operation engine is expanded to include an entry for a `data store location` (2 bits). The input selector is enhanced to allow the data store as a source.

Third, the configuration-store is augmented to hold a `next-config` value for each configuration. This allows

processing for a single lookup to go through multiple configurations. When performing multi-cycle lookups, power is expended reconfiguring the datapath every cycle.

Finally, multi-cycle compute steps impact throughput. To sustain a throughput of one lookup every cycle, each tile must contain as many compute engines as C pipeline stages. If a lookup requires multiple SRAM reads, throughput will be degraded, or the SRAM must allow multiple read ports. Formally, the total throughput in terms of cycles-per-lookup can be defined as: $\max(\frac{\#\text{ compute-steps}}{\#\text{ compute-engines}}, \frac{\#\text{ sram-reads}}{\#\text{ sram-ports}})$

As part of the compilation, the compiler does this analysis and can inform the system software of the cycles-per-lookup which can then be used in higher level scheduling of lookups to the lookup engine.

Although this multicycle support is not currently integrated into the actual LEAP prototype, it is presented here for future work.

5. LEAP COMPUTE ENGINE MICROARCHITECTURE

This section details the microarchitectural aspects of LEAP and is meant to be a reference for implementing in RTL a working LEAP design. Section 4 details the high level view of the architecture but this section goes into the detailed implementation of LEAP. For example, Figure 4(e) represents a generalized input selector for a generic operation engine but this section details the input selection for the operation engines chosen in Section 4.3. The SRAM address and data formation and the output network messages, details left for the implementers, are also specifically detailed in this section.

When a network message arrives (or a set of network messages, in which case they must all have the same code block), the codeblock of the message is used to index into each of the distributed configuration stores (one for each operation engine, one for the read stage and one for the write stage). These configuration bits configure the datapath as programmed to achieve the desired computation. Table 4 shows the number of configuration bits required for each pipeline stage. Configuration sizes vary between stages and between operation engines because each has unique functionality. Below we describe the exact function of the configuration bits for each stage and operation engine by describing the microarchitecture of LEAP. The description is broken down by pipeline stage and within a stage is broken down into that stage's various components.

5.1 Read Stage

The read stage is computationally simple. A 16-bit address is generated by first selecting a 16-bit subset of any of the input messages. This 16-bit address is then fed to an arbitrary right shifter as shown in Figure 8. If 32 bit addresses are needed, this shifted lower 16-bits are augmented with an additional mux to select the upper 16-bits. Configuration

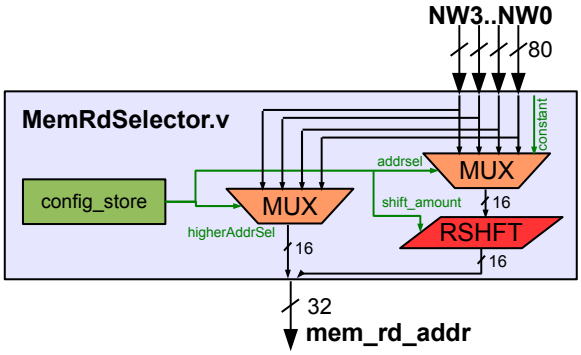


Figure 8: Read Address Formation

bits (shown in green) provide the selection bits for the upper and lower 16 bits, a constant for a constant address, and the amount for the right shifter to shift.

In initial prototypes of LEAP, address generation could borrow operation engines from the compute stage and use them to calculate an address. However, this required bypass logic that would feed Read stage values into the compute stage for processing. Rather than incur such an overhead, the most recent implementation of LEAP adds any computational power required to generate addresses into the Read stage. If future applications require more robust address formation, the Read stage is likely to grow in its area footprint. However, since LEAP's compute engine is small in power and area relative to the memory in each tile, this tradeoff is justified.

For packet classification, the SRAM could be *logically* enhanced to handle strided access. The config store sequences the SRAM so one 128-bit value is treated as multiple addresses. The only enhancement to the SRAM is an added external buffer to hold the 128 bits and logic to select a subset based on configuration signals. The current version of LEAP does not yet have this support.

5.2 Compute Stage

The computational operations of LEAP occur in the compute stage in both opClusters. The opClusters are fed the same inputs but only one data value from each cluster's operation engines can be fed to the operation engines in the other cluster. Each cluster has its own configuration store that can set the output of that cluster's link to any of its eight operation engines.

Computation is done by chaining all the operation engines in one clock cycle. The longest path (through 8 operation engines in one cluster, across the link and through the remaining 8 in the other cluster) is not intended to fit in one cycle and the compiler either needs to know the longest allowed path or have a way to slow the frequency of the LEAP chip. The functions of the operation engines are described below.

Add/Sub Operation Engine The Add/Sub operation engine is a 32 bit adder/subtractor with an additional optional decre-

Pipeline Stage		*Total Config Bits*
Read		**33**
Compute		**890**
OpCluster (2 per Engine)		*445*
	Add	46
	Mux2	47
	Logic2	48
	Logic4 (2 per Cluster)	66
	BitCount	18
	BSPTree	131
	Bitsel	19
	ClusterLink	4
Write		**998**
Writeback		*82*
Message Formation (4 per Engine)		*229*
LEAP Tile		***1921***

Table 4: Number of configuration bits by pipeline stage and by operation engine

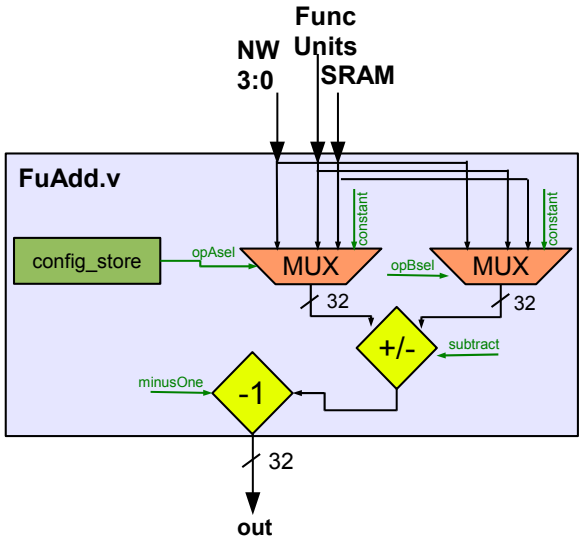


Figure 9: Add/Sub Operation Engine

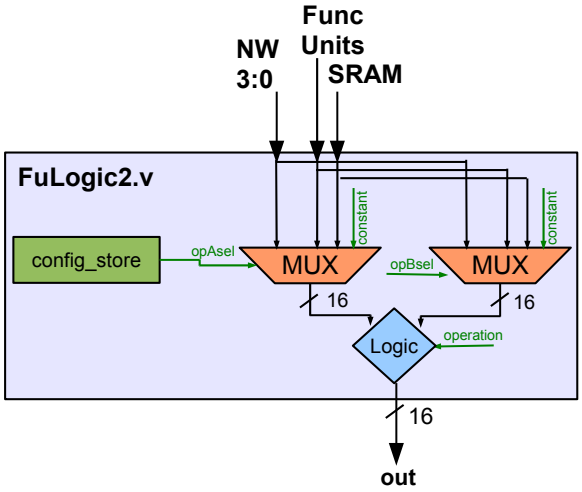


Figure 10: Logic2 Functional Unit

ment. See Figure 9 for a visual representation. The operands are selected by *opASel* and *opBSel* in the configuration bits stored in the configuration store and either can be set to a *constant* from the configuration store or any aligned 32 bits from the SRAM, network messages or other operation engines.

If *subtract* is set the unit subtracts B from A otherwise it adds. *MinusOne* when set, subtracts one off the previous result before outputting.

Logic2 Functional Unit

The Logic2 unit performs a logical operation on two operands as shown by Figure 10. The 16 bit operands are selected by *opASel* and *opBSel* and either can be set to a *constant* from the configuration store or any aligned 16 bits from the SRAM, network messages or other operation engines. The logical operation can be a logical OR, AND, XOR, left shift,

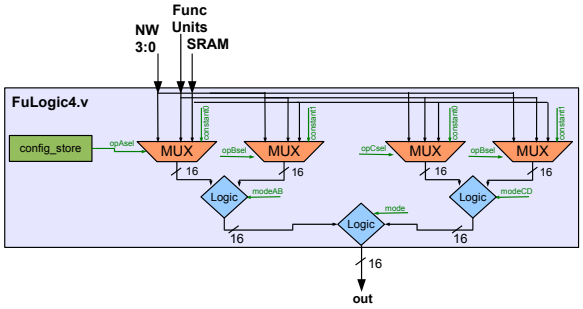


Figure 11: Logic4 Operation Engine

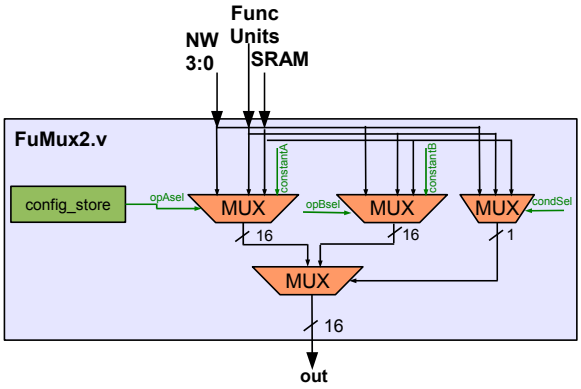


Figure 12: Mux2 Operation Engine

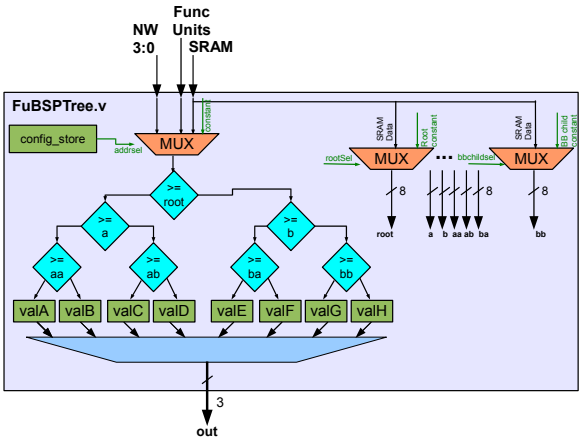


Figure 13: Binary Search Tree Operation Engine

or right shift. Additionally, the operation can be an unsigned greater than, greater than or equal, or equal comparison. The result of this comparison, if true, results in an output of 1. The comparison output can be flipped (1 if false) by setting the *invert* option in the configuration. The operation, either a bitwise logical operation or a comparison option, is set by *operation*.

Logic4 Operation Engines The two Logic4 units in each cluster act as two Logic2 units being fed into a third Logic2 unit. Figure 11 shows the normal operation. *OpASel*, *opBSel*, *opCSel*, and *opDSel* select the operands much like in Logic2. The constant fed into the operand mux is shared between A and C as well as B and D. *ModeAB* and *modeCD* choose between the same operations as in Logic2. However, the left shift operation is reversed, with operand A being the shift amount and operand B being the value to shift. This was done to allow the Logic4 to both do a left shift and a right shift and then combine the results.

The output of the operations on the AB and CD pairs are then fed into a third logical operation block. This third block does either a logical OR, AND, XOR, right shift, greater than comparison, greater than or equal to comparison, or an equal comparison as set by *mode* in the configuration. Additionally *mode* can be set to *SEL* to select between operandC and operandD based on the LSB of the output from the A and B logic unit; if the operation on A and B results in a 1 in the LSB operandD is output, otherwise operandC is output.

Mux2 Operation Engine Figure 12 shows the Mux2 operation engine. It is used to choose between two 16-bit values (A and B). The values of A and B are chosen by setting the configuration of *opASel* and *opBSel*. A and B can both be either their own constant (set by *constantA* and *constantB*), part of the network messages, a value from SRAM or a value from another operation engine in the cluster (or from the other cluster through the cluster link).

The *condSel* selects a source for the condition. The LSB of the selected condition chooses operandA if 0 or operandB if 1. Note that the SEL operation of Logic4 can also achieve the same functionality as a Mux2.

Binary Search Operation Engine The binary search tree takes a value and sorts it using a 3-deep tree. Each node of the tree does a less than operation to see if the search value is less than the node. If the value is greater or equal than the node the tree is traversed to the right child, otherwise it goes to the left child. Since it is unsigned, a zero value at a node results in the tree always being traversed to the right. At the leaves, the value output is chosen from 3 bit constants from the configuration store (*valA..valH*). A visual representation of this is shown in Figure 13.

Much like other operation engines, the 8 bit value used to traverse the tree is selected by *opASel* and can be from a *constant*, the SRAM, network messages or other operation engines. The 8 bit values at the nodes are chosen by *rootSel, achildSel, bchildSel, aachildSel, abchildSel, bachildSel,* and *bbchildSel*. These values can each be their own constant or from SRAM. They cannot be from other operation engines in this design.

Note that the BSPTree is the most specialized of the operation engines. It may change or expand its functionality in future versions of LEAP, specifically to act as an 8to1 MUX.

Bit Counter Operation Engine The Bit Counter opera-

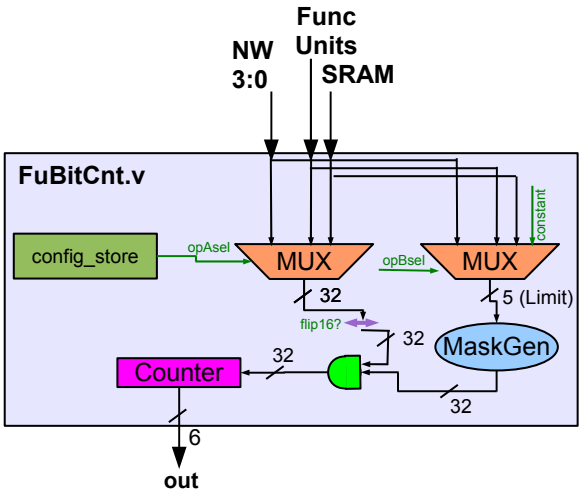


Figure 14: Bit Counter Operation Engine

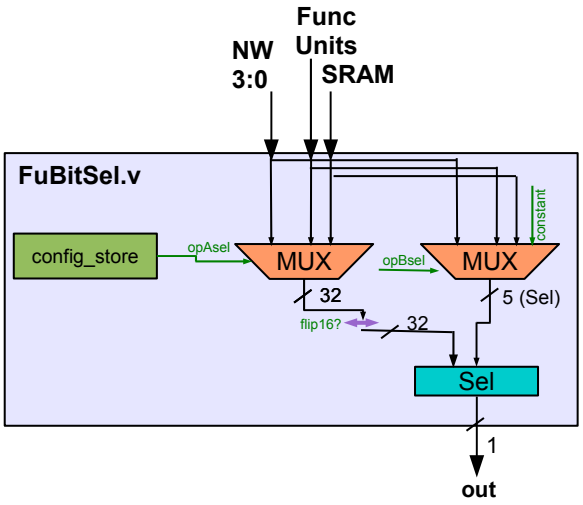


Figure 15: Bit Selector Operation Engine

tion engine is a masked bit counter. The 32-bit value to count is chosen by *opASel* as shown in Figure 14. The mask is generated by a 5-bit limit chosen by *opBSel*. This limit can be a *constant*. The limit can be treated as the number of bits minus one to count starting with the MSB counting down to the LSB. A limit of zero counts only the MSB and a limit of 31 counts all 32 bits. Optionally *flip16* can be set to flip the upper and lower 16 bits of the premasked value.

Bit Selector Operation Engine The Bit Selector selects a single bit from a 32-bit operand chosen by *opASel*. *OpBSel* chooses the value with which to index into operandA and choose a bit. The indexed value can be a *constant* or be from SRAM, a network message, or another operation engine. Likewise the operandA to select from can be from SRAM, the network or an operation engine. Optionally, the upper and lower 16 bits of operandA can be flipped by setting *flip16*. See Figure 15.

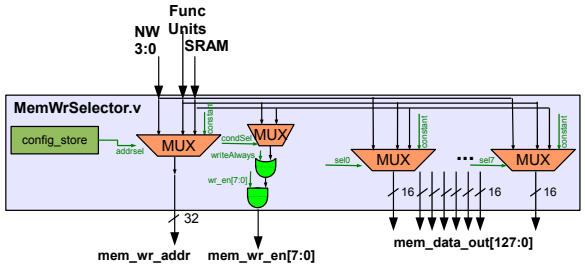


Figure 16: Write Stage

5.3 Write/Message Formation Stage (Bit Collector)

The final pipeline stage both writes back to the SRAM and forms up to four output messages. This stage is represented in the high-level LEAP architecture by the Bit Collector unit.

SRAM Write The SRAM needs to have simultaneous write and read access so the writeback does not conflict with the read stage. If a two-port SRAM is not used and write is not a common task, appropriate bubbles could be inserted manually or by the scheduler. The initial design assumes a dual-port SRAM (which is supported on most modern FPGAs).

The address can be selected from a read SRAM value, a network message, a *constant* or output from either cluster of operation engines. The address is selected by setting *addrSel* in the configuration. Data values can be selected from the same sources in aligned 16-bit granularities by setting the corresponding *sel0* through *sel7* configuration values.

Enabling the writeback is achieved in two steps as shown by Figure 16. First the appropriate 16-bit chunks to write is chosen by setting *wr_en[7:0]*. If a write should always occur for a given codeblock, *writeAlways* should be set to 1. Otherwise, *condSel* can be used to select one of the logical operation engines' outputs and if the LSB of the output is 1 a write occurs for all chosen 16-bit chunks. To disable all writes, *wr_en[7:0]* simply needs to be set to all zeros.

Network Message Formation Also contained in the write stage is the network message formation. Much like the operation engines and the memory read/write units, the bit collector that forms network messages has its own configuration store. Figure 17 shows one of four network message units. Values in the network message can be assembled in 8-bit granularities and can be taken from the input network message, the SRAM and the output of operation engines in both clusters.

If *SendAlways* is set, a network message is always produced. If *SendConditional* is set, the message is sent when the LSB of the selected condition is 1. The condition is selected by *selCond*. Alternatively, if the configuration has *ConditionalReduction* set, instead of the LSB, a reduction AND of the condition is done instead of just taking the LSB

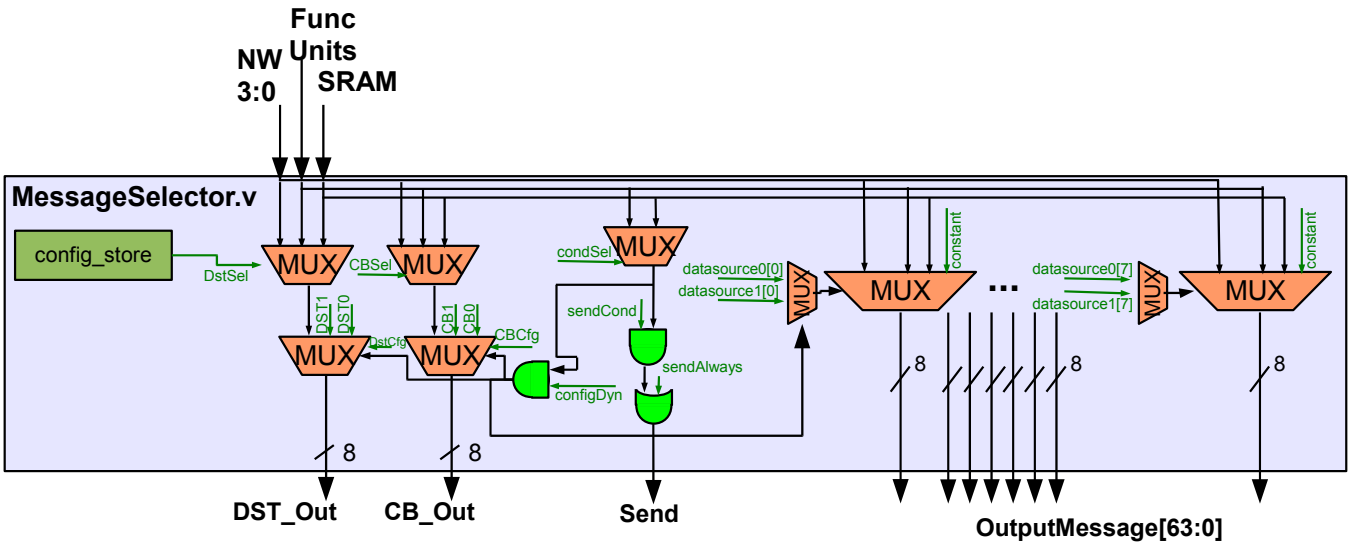


Figure 17: Output Message Formation for one network

(not shown in Figure 17).

The Message Selector, which is the unit that forms the network message for one network, can store two different configurations on how to form the network message. This includes two desintations, two target codeblocks and two message data sources. If *ConfigDyn* is set in the configuration vector, the same condition as selected by *selCond* (and optionally reduction AND'ed by *ConditionalReduction* being set) is used to choose between the two configurations for forming the message. Thus *DST0* and *DST1* set the two target destinations and *CB0* and *CB1* choose the two target codeblocks. If *ConfigDyn* is not set, then the first (*DST0, CB0*) configuration is always chosen. Alternative to choosing between two constant destinations and two constant codeblocks *DstConfig* and *CBConfig* can be set. If either of these is set, then for either the destination or codeblock source, *DstSel* or *CBSel* respectively select a source to use as the target destination or codeblock.

Like there are two configurations for destination and codeblock selection, there are two configurations for the 64 bit data payload of the network message. If *ConfigDyn* is set then the same conditional selects between the two, otherwise the first is used. *Datasource0[0]* and *datasource1[0]* select the source for the LSB byte of the payload and *datasource0[7]* and *datasource1[7]* select the highest byte of the data message. Data sources selected by these can be from either cluster's operation engines, SRAM or input network messages. They also can be from an array of constants, shown in Figure 17 by the green *constant* input to each mux. These constants can be different for each byte selector but are shared between the first and second configurations.

6. IMPLEMENTATION AND VERIFICATION

6.1 RTL Implementation in Verilog

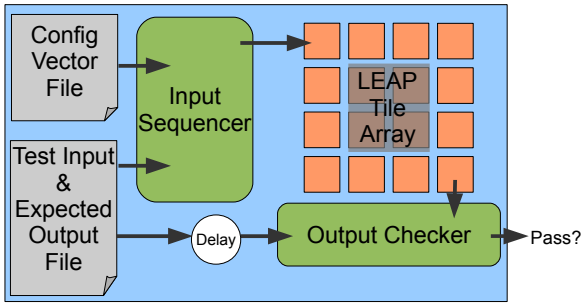


Figure 18: Simulation Verification Setup

The microarchitecture previously presented was implemented in 2001 Verilog RTL. The RTL was designed with behavioral logic but only synthesizable syntaxes. The RTL design was modular and modules mirrored the compute engine, operation cluster, operation engine hierarchy previously presented.

6.2 Verification Setup

This section describes the steps taken and the framework used to verify the implementation of the LEAP architecture described in the previous sections.

Verification is done in simulation with an automated testbench. The simulation setup is shown in Figure 18. The automated testbench has three parameters that are modified for each test at the top of the testbench file. First, a user must specify the path to a file with the configuration vectors. The configuration file must have vectors of the following format:

```
DST0 CB0 DAT64_48 DAT47_32 DAT31_16 DAT15_0
```

Each line must have a vector for all four networks, separated by a space. X's can be used to indicate the network is not used.

The testbench must also be provided a test file. The test file follows a similar format but each line has a space-less string before the first input for the first network, inputs for 4 networks, the string "EXPECT" and then four output messages for the expected output.

The final testbench parameter is the expected latency from when the input is inserted and when the output is expected. If only a single LEAP tile is being tested this latency would be three for the three pipeline stages. As part of this step, the correct size LEAP tile array has to be instantiated.

The testbench is limited in that if the SRAMs of a tile need to be initialized, it can currently only initialize all tiles to the same SRAM contents. Additionally, the testbench can only run one test at a time.

Verification Helper Scripts There are a few helper scripts written to help format and prepare the test files.The file **rtl/leap/tests/formatter.cpp** helps reformat test files written for PLUG to a format that works for LEAP. The file **rtl/leap/LEAP_Array_Generator.cgi** generates LEAP tile arrays of arbitrary sizes and makes the proper connections between them.

7. PHYSICAL DESIGN CONSIDERATIONS

This section details some concerns of physical synthesis and how they affect LEAP and may change LEAP in the future.

7.1 Combinational Loop Resolution

LEAP has combinational loops by design. Since each functional unit can take input from the other function units, configurations which create loops, while undefined in operation, are possible. These loops are detected by the synthesis tools and arbitrarily broken. For initial synthesis timing purposes, latches that run at a much higher frequency than the core clock were inserted at the output of each functional unit. These latches are not mean to be realized in the final taped-out design.

An alternate and more permanent solution is to use synthesis rules to set the configuration bits in the configuration stores to tell synthesis no combinational loops are present. The bits are set to the longest expected critical path (or longest allowed critical path) and synthesis can proceed that way. This solution is still being investigated. To help create the synthesis rules setting the configuration bits, a system was built to set the bits. The system duplicates much of the functionality of the Java programming aid described in Section 9. In the future, these two systems are expected to be merged.

7.2 Fanout due to Configuration

LEAP was initially designed to reuse the existing inter-tile networks for functional unit configuration loading. However, feeding the network inputs to every configuration store for each functional unit greatly increases the fanout and affects the critical path. In the future, LEAP may use a JTAG or JTAG-like interface for configuration loading.

Algorithm	Critical Path
Ethernet Forwarding	Logic4→Logic4
IPv4 forwarding	Logic2→Link→Logic4→Mux2→Add→Logic4
	Logic2→BSPTree→Add→Logic4
IPv6 forwarding*	Logic4→Logic2→Logic2→Mux2
	BitCnt→Add→Mux2
	BSPTr→Add→Mux2
Packet Classification*	BitSel→Logic4→Mux2→Mux2→Add
	Mux2→Logic4→Add→BitSel
	Logic4→Logic→Mux2
DFA lookup*	BSPTr→BitSel→Add→BitSel
	Logic2
Ethane*	Logic4→Logic4→Logic→Mux2
SEATTLE*	Logic4→Logic2→Mux2

*=unvalidated

Table 5: Examples of processing steps' critical paths.

8. DISCUSSION OF TRADEOFFS

Functional unit mix: Based on the analysis in Section 3.1, we determined an appropriate functional-unit mix by implementing specialized hardware designs in Verilog (details in Section 10). This showed that various lookups use a different mix of a core set of operations, justifying a dynamically synthesized lookup engine. Table 5 presents a sample across different applications showing the use of different operation engines by listing the critical path in terms of functional units serially processed in a single compute step.

Bit selection: From the fixed-function implementation, we observed that a commonly used primitive was to select a subset of bits produced by a previous operation. In a programmable processor like PLUG this is accomplished using a sequence of shifts and or's, which uses valuable cycles and energy. To overcome these overheads, every functional unit is preceded by a bit-selector which can select a set of bits from the input, and sign- or zero- extend it. This is similar to the shift mechanisms in the ARM instruction sets [1].

Crossbar design: Instead of designing a "homogenous" crossbar that forwards 16 bits across all operation engines, we designed one that provides only the required number of bits based on the different functional units. For example *bitcount*, *bitselect*, and *bsptree-search* produce 5 bits, 1 bit, and 4 bits of output respectively. This produces savings in latency and area of the crossbar.

A second piece of the crossbar's unusual design is that its critical path dynamically changes based on the configuration. We have verified through static timing analysis that any four serial operations can be performed in a single cycle. This would change if our mix of functional units changed.

Scalability: A fundamental question for the principles on which LEAP is constructed is what ultimately limits the latency, throughput, area, and power. This is a sophisticated multi-way tradeoff, denoted in a simplified way in Figure 19. With more area, more operation engines can be integrated into a compute engine. However, this will increase the la-

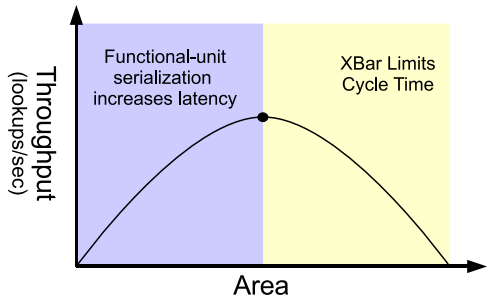


Figure 19: Design scalability tradeoff

tency of the crossbar, thus reducing frequency and throughput. If the area is reduced, then specialized units like the *bsptree-search* must be eliminated and their work must be accomplished with primitive operations, increasing latency (and reducing throughput). A faster clock speed cannot make up the processing power lost because the cycle time is lower-bounded by the SRAM. Increasing or reducing power will cause a similar effect. For the architecture here, we have proposed an *optimized* and *balanced* design for a target throughput of 1 billion lookups-per-second and overall SRAM size of 256KB (split across four 64KB banks). With a different target, the type and number of elements would be different.

9. PROGRAMMING LEAP

This section describes the programmer's abstract machine model view of LEAP, describes a specific Python API to program LEAP, discusses current compilation tools and outlines an example in detail showing final translation to LEAP configuration bits. The API provides a familiar model to programmers despite our unique microarchitecture.

9.1 Abstract machine model

The abstract machine model of LEAP hides the underlying concurrency in the hardware. Specifically, the programmer assumes a serial machine that can perform one operation at a time. The only types of operations allowed are those implemented by the operation engines in the hardware. The source for all operators is either a network message, a value read from memory, or the result of another operator. The native data-type for all operators is bit-vector and bit range selection is a primitive supported in the hardware. For example, `a[13:16]` selects bits 13 through 16 in the variable `a`, and comes at no cost in terms of latency.

This machine model, while simple, has limitations and cannot express some constructs. There is no register file, program counter, control-flow, stack, subroutines or recursion. While this may seem restrictive, in practice we found these features unnecessary for expressing lookups.

Lack of control-flow may appear to be a significant limitation, but this is a common choice in specialized architectures. For example, GPU programming did not allow support for arbitrary control flow until DirectX 8 in 2001. The LEAP machine model does allow two forms of conditional execution. Operations that alter machine state – stores and message sends – can be executed conditionally depending on the value of one of their inputs. Also, a special select primitive can dynamically pick a value from a set of possible ones, and return it as output. Both capabilities are natively supported by LEAP, and we found that they were flexible enough to implement every processing step we considered.

9.2 Python API for LEAP

We have developed a simple Python API with the goal of making programming LEAP practical. In our model, programmers express their computational steps as Python subroutines using this API.

Given the simplicity of LEAP units and the abundance of functionality (e.g. bitvectors) in Python, a software-only implementation of LEAP API calls is trivial. Developers simply run the code on a standard Python interpreter to verify syntactic and semantic correctness. After this debugging phase, a compiler converts this Python code into binary code for the configuration store.

The functionality provided by every functional unit is specified as a Python subroutine. An entire compute step is specified as a sequential set of such calls. Recall the compiler will extract the concurrency and map to hardware. Table 6 describes the most important calls in the API. In most cases a call is mapped directly to a LEAP functional unit; some calls can be mapped to multiple units, for example if the operands are larger than a word. The common case of a comparison between two long bitvectors is optimized through the use of the LOGIC4 unit, which can perform two partial comparisons, and AND them together.

9.3 Compiler and Java Hardware Model

LEAP is intended to be programmed in the Python API and then handled to a compiler that will map API calls to the hardware. If a written program will not fit on an instance of LEAP hardware because it requires more hardware resources, the compiler can either inform the programmer or break the program into a subportion that will map to hardware. For the implementation of the LEAP architecture (what was described in section 5) there is not yet a compiler.

Java Hardware Model Though there is no compiler for LEAP, a model of the LEAP was designed in Java. The model contains objects with properties that correspond to hardware and their possible configurations. This model's objects are meant to be similar to nodes in an directed acyclic graph (DAG) that a compiler would map the API calls to. For a given instance of a Tile object, the object model's properties exactly corresponds to the configurations for the LEAP hardware. The Java model has the added advantage of being able to generate the configuration vectors required for the LEAP hardware and was used to help develop and verify applications on the LEAP.

Figure 20 shows the previous Ethernet forwarding example in both the Python API and the Java object configuration corresponding to the Python API calls. The calls in Python

API Call	Functional unit	Description
Access functions		
read_mem(addr)	SRAM	Load value from memory
write_mem (addr)	SRAM	Store value to memory
send (value, enable)	Network	Send `value` on a on-chip network if `enable` is not 0.
Operator functions		
select (v0, v1, sel, width=16—32—64)	Mux2	Selects either `v0` or `v1` depending on the value of `sel`
copy_bits (val[a:b])	BitSel	Extracts bits between position `a` and position `b` from `val`
bitwise_and(a, b)	Logic2	bitwise AND
bitwise_or(a, b)	Logic2	bitwise OR
bitwise_xor(a, b)	Logic2	bitwise XOR
eq(a, b)	Logic2	Comparison (returns 1 if `a == b`, 0 otherwise)
cond_select(a, b, c, d, "logic-function")	Logic4	Apply the logic-function to `a` and `b` and select `c` or `d` based on the result.
add(a, b)	Add	Sum `a` to b
add_dec(a, b)	Add	Sum `a` to `b` and subtracts 1 to the result
sub(a, b)	Add	Subtract `b` from `a`
bitcount(value, start_position)	BitCnt	Sum bits in `value` from bit `start_position` to the end
bsptree3_short(value, vector, cfg, res)	BSP-Tree	Perform a binary-space-tree search on vector. `value` is an 8-bit value; `vector` is a 64-bit vector including 7 elements, each 8 bits; `cfg` is an 8-bit configuration word (1 enable bit for each node) `res` is a 24-bit value consisting of 8 result fields, each 3 bits wide.
bsptree3_long(value, vector, cfg, res)	BSP-Tree	Perform a binary-space-tree search on 128-bit vector with 16-bit value.

Table 6: Python API for programming LEAP

roughly correspond to configuration settings in the Java. For example, the comparison in line 4 of the Python is a 48 bit comparison. To map this to the hardware, two Logic4 units are set up for an equality comparison in the Java code lines 5 through 22. Line 7 in the Python, which shows a conditional send, corresponds to setting sendConditional to True in Java on line 28 and setting the condition to the equality mapped by the Logic4 units as shown in Java line 29. In the future, a compiler would do this mapping. The Java model can easily spit out binary configuration bits corresponding to the proper configuration. These bits are not shown in the figure because the object configuration shown exactly corresponds to the non-human readable configuration vector format.

10. EVALUATION

To quantitatively evaluate LEAP, it is compared to an optimistic model constructed for a fixed-function lookup engine for each lookup (see Appendix A. LEAP is also compared to PLUG which is a state-of-art programmable lookup engine. Note that numbers in this section may not be representative of the very latest RTL but can be treated as an estimate. These numbers are as submitted to ANCS.

10.1 Methodology

The seven algorithms mentioned in Table 1 were implemented using the Python API including the multiple algorithmic stages and the associated processing. For each lookup, additional data like a network traffic trace or lookup trace and other dataset information is used to populate the lookup data structures. These vary for each lookup and Table 7 describes the data sets used. To be consistent with the quantitative comparison to PLUG,similar or equivalent traces and datasets were picked. For performance of the hardware we consider parameters from our RTL prototype implementation: clock frequency is 1 GHz and we used the 55nm Synopsys synthesis results to determine how many compute steps

Algorithm	Lookup data
Ethernet forwarding	Set of 100K random addresses
IPv4 forwarding	280K-prefix routing table
IPv6 forwarding	Synthetic routing table [41]
Packet classification	Classbench generated classifiers [34]
DFA lookup	Signature set from Cisco [2]
Ethane	Synthetic data based on specs [11]
SEATTLE	Synthetic data based on specs [25]

Table 7: Datasets used

lookup processing took for each processing step at each algorithmic stage.

Modeling of other architectures: To determine how close LEAP comes to a specialized fixed-function lookup engine (referred to as FxFu henceforth), we would like to consider performance of a FxFu hardware RTL implementation. Recall that the FxFu is also combined with an SRAM like in PLUG and LEAP. We implemented them for three lookups to first determine whether such a detailed implementation was necessary. After implementing FxFu's for Ethernet forwarding, IPv4, and Ethane, we found that the they easily operated within 1 ns, consumed *less than 2%* of the tile's area, and the contribution of processing to power consumption was always less than 30%. Since such a level of RTL implementation is tedious and ultimately the FxFu's contribution compared to the memory is small, we did not pursue detailed fixed-function implementations for other lookups and adopted a simple optimistic model: we assume that processing area is fixed at 3% of SRAM area, power is fixed at 30% of total power, and latency is always 2 cycles per-tile (1 for memory-read, 1 for processing) plus 1 cycle between tiles.

We also compare our results to the PLUG design by considering their reported results in [26] which includes simulation- and RTL-based results for area, power, and latency. For all three designs we consider a tile with four 64KB memory banks. With 16 total tiles, we can get 4MB of storage thus providing sufficient storage for all of the lookups.

Metrics: We evaluate latency per lookup, worst-case total

```
def cb1(): #Lookup Codeblock
 data = read_mem(nw0.data[15:0])
 # Get memory value from input bits 0-15
 ifCheck = EQ(data[47:0],nw0.data[63:16])
 # Compare loaded memory val w/ input bits 16-63
 #
 nw0_out.send = ifCheck
 # Set valid bit in outgoing message header conditionally
 nw0_out.DST = const(0xFF)
 # Set destination page coordinates
 nw0_out.CB = const(0x00)
  #Set CB to zero
 nw0_out.data[63:32] = const32(0x00000000)
 # Set upper 32 bits of out message to 0
 nw0_out.data[31:16] = data[63:48]
 # Memory bits 48-63 become 16-31 in outgoing msg
 nw0_out.data[15:0] = const16(0)
```

(a) Python code

```
//Mem read
fwd[1].memrd.active = true;
fwd[1].memrd.addrSource = GlobalOperand16.nw0_15_0;

//48 bit comparison (All done on cluster 0)
fwd[1].op0.fulogic4_0.active = true;
fwd[1].op0.fulogic4_0.a = Operand16.OP_NW0_31_16;
fwd[1].op0.fulogic4_0.b = Operand16.OP_MEM15_0;
fwd[1].op0.fulogic4_0.c = Operand16.OP_NW0_47_32;
fwd[1].op0.fulogic4_0.d = Operand16.OP_MEM31_16;
fwd[1].op0.fulogic4_0.modeAB = Logic2Modes.MD_EQ;
fwd[1].op0.fulogic4_0.modeCD = Logic2Modes.MD_EQ;
fwd[1].op0.fulogic4_0.mode = Logic4Modes.MD_AND;

fwd[1].op0.fulogic4_1.active = true;
fwd[1].op0.fulogic4_1.a = Operand16.OP_NW0_63_48;
fwd[1].op0.fulogic4_1.b = Operand16.OP_MEM47_32;
fwd[1].op0.fulogic4_1.c = Operand16.OP_LOGIC4_0;
fwd[1].op0.fulogic4_1.d = Operand16.OP_LOGIC4_0;
fwd[1].op0.fulogic4_1.modeAB = Logic2Modes.MD_EQ;
fwd[1].op0.fulogic4_1.modeCD = Logic2Modes.MD_OR;
fwd[1].op0.fulogic4_1.mode = Logic4Modes.MD_AND;

//OutMessage Formation
fwd[1].bc.messageSels[0].dest0 = (byte)0xFF;
fwd[1].bc.messageSels[0].cb0 = (byte) 0x01;
fwd[1].bc.messageSels[0].sendAlways = false;
fwd[1].bc.messageSels[0].sendConditional = true;
fwd[1].bc.messageSels[0].conditional =
                    GlobalOperand1.BL_0_LOGIC4_1;
fwd[1].bc.messageSels[0].configDyn = false;
fwd[1].bc.messageSels[0].DstConfig = false; //(0)
fwd[1].bc.messageSels[0].CBConfig = false; //(0)
GlobalOperand8[] data = {GlobalOperand8.BC_CONST,
 GlobalOperand8.BC_CONST,GlobalOperand8.BC_MEM55_48,
 GlobalOperand8.BC_MEM63_56,GlobalOperand8.BC_CONST,
 GlobalOperand8.BC_CONST,GlobalOperand8.BC_CONST,
 GlobalOperand8.BC_CONST};
byte[] constants = {0,0,0,0,0,0,0,0};
fwd[1].bc.messageSels[0].dataSource0 = data;
fwd[1].bc.messageSels[0].constants = constants;
```

(b) LEAP configuration bits in human readable Java code

Figure 20: Ethernet forwarding example compute step

Algorithm	Total Algorithmic Stages	Total Compute Steps	Avg. Lines per Compute Step
Ethernet forwarding	2	6	9.5
IPv4	8	42	10.8
IPv6	26	111	12.1
Packet classification	3	3	98
DFA matching	3	7	9.5
Ethane	5	22	11.5
SEATTLE	4	19	9.3

Table 8: Application Code Statistics

power (dynamic + static), and area of a single tile. Chip area is tile area multiplied by the number of tiles available on the chip plus additional wiring overheads, area of IO pads, etc. The fixed-function engines may be able to exploit another source of specialization in that the SRAM in tiles can be sized to exactly match the application. This requires careful tuning of the physical SRAM sub-banking architecture when algorithmic stage sizes are large along with a design library that supports arbitrary memory sizes. We avoid this issue by assuming FxFu's also have fixed SRAM size of 256 KBs in every tile. Finally, when SRAM sizes are smaller than 64KB, modeling tools like CACTI [35] overestimate. Our estimate of the FxFu area could be conservative since it does not account for this memory specialization.

10.2 Implementing Lookups

First, we demonstrate that LEAP is able to flexibly support various different lookups. Table 8 summarizes code statistics to demonstrate the effectiveness of the Python API and ease of development for the LEAP architecture. As shown in the second and third columns, these applications are relatively sophisticated, require accesses to multiple memories and perform many different types of processing tasks. The fourth column shows that all these algorithmic stages can be succinctly expressed in a few lines of code using our Python API. This shows our API provides a simple and high-level abstraction for high-productive programming. All algorithmic stages in all applications except Packet classification are ultimately transformed into single-cycle compute steps.

Result-1: LEAP and its programming API and abstraction are capable of effectively implementing various lookups.

10.3 Performance Analysis

Tables 9-11 compare the fixed-function optimistic engine (FxFu), PLUG and LEAP along the three metrics. All three designs execute at a 1 GHz clock frequency and hence have a throughput of 1 billion lookups per second on all applications except Packet-classification.

Latency: Table 9 shows latency estimates. For FxFu, latency in every tile is the number of SRAM accesses plus one cycle of compute plus one cycle to send. Total latency is always equal to the tile latency multiplied by number of tiles accessed. For LEAP, all lookup steps except Packet classification map to one 1ns compute stage. The laten-

Algorithm	FxFu	PLUG	LEAP
Ethernet forwarding	6	18	6
IPv4 forwarding	24	90	24
IPv6 forwarding	42	219	42
Packet classification	23	130	75
DFA matching	6	37	6
Ethane	6	39	6
SEATTLE	9	57	9

Table 9: Latency Estimates (ns)

	FxFu	PLUG	LEAP
Total	37 mWatts	63 mWatts	49 mWatts
Memory %	70	42	54
Compute %	30	58	46

Table 10: Power Estimates

	FxFu	PLUG	LEAP
Total	2.0 mm^2	3.2 mm^2	2.1 mm^2
Memory %	97	64	95
Compute %	3	36	5

Table 11: Area Estimates

cies for PLUG are from reported results. For FxFu and LEAP the latencies are identical for all cases except Packet-classification since compute-steps are single cycle in both architectures. The large difference in packet classification is because our FxFu estimate is quite optimistic - we assume all sophisticated processing (over 400 lines of C++ code) can be done in one cycle, with little area or energy. For PLUG, the latencies are universally much larger, typically on the order of $5\times$ larger, for two reasons. First, due to its register-file based von-Neumann design, PLUG spends many instructions simply assembling bits read-from/written-to the network. It also uses many instructions to perform operations like bit-selection which are embedded into each operation engine in LEAP. A second and less important factor is that LEAP includes the *bsptree-search* unit that is absent in PLUG.

Result-2: LEAP matches the latency of fixed-function lookups and outperforms PLUG by typically $5\times$.

Energy/Power: Since all architectures operate at the same throughput, energy and power are linearly related; we present our results in terms of power. For FxFu and LEAP, we estimate power based on the results from RTL synthesis and the power report from Synopsys Power Compiler, assuming its default activity factors. For PLUG, we consider previously reported results also at 55nm technology. Peak power of a single tile and the contribution from memory and processing are shown in Table 10.

Result-3: LEAP is $1.3\times$ *better than PLUG in overall energy efficiency. In terms of processing alone, LEAP is* $1.6\times$ *better.*

Result-4: Fixed-function designs are a further $1.3\times$ *better than LEAP, suggesting there is still room for improvements.*

Area: We determined tile area for FxFu and LEAP from our synthesis results and use previously reported results for PLUG. These are shown in Table 11. The network and router area is small and is folded into the memory percentage.

Result-5: LEAP is $1.5\times$ *more area efficient than PLUG overall. In terms of processing area alone, it is* $9.4\times$ *better.*

Result-6: LEAP is within 5% of the area-efficiency of fixed-function engines, overall.

11. FUTURE WORK

The LEAP architecture and its implementation is an ongoing research topic being pursued by the Vertical Research Group at UW-Madison. This section describes the anticipated future work and the suggested next steps for the next researcher(s).

The current prototype only has two working applications: Ethernet forwarding and IPv4. Implementing and verifying more applications (for which there are already portions of the Python API implementation) on the LEAP implementation is a likely next step. There is already a test framework present for simulation-based verification so this step is ready to be worked on.

FPGA prototyping is also another likely next step. LEAP was created with FPGA prototyping in mind. The FPGA test framework is lacking and this is another likely next step for development.

The physical design aspects of LEAP are currently being investigated. It is likely they will lead to changes to LEAP in the near future. This is a third task for future work.

Additionally as part of this project, the LEAP architecture was drafted into a patent in cooperation with the Wisconsin Alumni Research Foundation (WARF) and filed. The result of this application may change or direct future work on LEAP.

For future researchers, the SVN of the LEAP project has been tagged with *LEAPSummer2012*. The SVN contains copies of this report, the RTL for LEAP and all application implementations to date for LEAP. The java model is contained in compiler folder of the SVN. It also includes the fixed function implementations of the applications. Permanent contact information for Eric Harris is listed at this top of this report.

12. CONCLUSION

Data plane processing in high-speed routers and switches has come to rely on specialized lookup engines for packet classification and various forwarding lookups. In the future, flexibility and high performance are required from the networking perspective and improvements in architectural energy efficiency are required from the technology perspective.

LEAP presents a tractable path to efficient soft-synthesizable multifunction lookup engines. By using a dynamically configurable data path relying on coarse-grained functional units, LEAP avoids the inherent overheads of von-Neumann-style programmable modules. Through our analysis of several lookup algorithms, we arrived at a design based on 16 instances of seven different functional units together with the required interconnection network and ports connecting to the memory and on-chip network. Comparing to PLUG, a state-of-art flexible lookup engine, the LEAP architecture offers the same throughput, supports all the algorithms im-

plemented on PLUG, and reduces the overall area of the lookup engine by $1.5\times$, power and energy consumption by $1.3\times$, and latency by typically $5\times$. A simple programming API enables the development and deployment of new lookup algorithms. These results are comprehensive, promising, and show the approach has merit.

A complete prototype implementation with an ASIC chip or FPGA that runs protocols on real live-traffic is on-going and future work to demonstrate LEAP's quantitative impact in product and deployment scenarios. By providing a cost-efficient way of building programmable lookup pipelines, LEAP may speed up scaling and innovation in high-speed wireline networks enabling yet-to-be-invented network features to move faster from the lab to the real network.

Acknowledgements

I'd like to thank fellow students Samual Wasmundt, Lorenzo De Carli and Sung Jin Kim for their significant work on LEAP. Cristian Estan and Ranga Sankaralingam also committed significant time in advising the development of LEAP. I'd like to thank Professor Karu Sankaralingam for the opportunity to learn and work on the LEAP project. For his knowledge, wisdom, dedication to research and my education, I am in his debt. Much of this report was based off SIGCOMM and ANCS submissions created with Sam, Lorenzo, Crstian and Karu so I'd like to thank them for helping create this report as well. Finally I'd like to thank Claude and Dora Richardson for funding my studies the last four years.

13. REFERENCES

[1] Arm instruction set reference https://silver.arm.com/download/download.tm?pv=1199137.

[2] Cisco intrusion prevention system. http://www.cisco.com/en/US/products/ps5729/Products_Sub_Category_Home.html.

[3] Cypress delivers industry's first single-chip algorithmic search engine. http://www.cypress.com/?rID=179, Feb. 2005.

[4] Neuron and neuronmax search processor families. http://www.cavium.com/processor_NEURON_NEURONMAX.html, Aug. 2011.

[5] F. Baboescu, D. Tullsen, G. Rosu, and S. Singh. A tree based router search engine architecture with single port memories. In *ISCA '05*.

[6] F. Baboescu and G. Varghese. Scalable packet classification. In *SIGCOMM '01*.

[7] A. Basu and G. Narlikar. Fast incremental updates for pipelined forwarding engines. In *IEEE INFOCOM '03*.

[8] F. Bonomi, M. Mitzenmacher, R. Panigraphy, S. Singh, and G. Varghese. Beyond Bloom filters: From approximate membership checks to approximate state machines. In *SIGCOMM '06*.

[9] S. Borkar and A. A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, 2011.

[10] A. Broder and M. Mitzenmacher. Using multiple hash functions to improve IP lookups. In *INFOCOM '01*.

[11] M. Casado, M. J. Freedman, J. Pettit, J. anying Luo, N. McKeown, and S. Shenker. Ethane: taking control of the enterprise. In *SIGCOMM '07*.

[12] 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.

[13] 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. In *SIGCOMM '09*.

[14] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. In *SIGCOMM '97*.

[15] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep packet inspection using parallel bloom filters. In *IEEE Micro*, pages 44–51, 2003.

[16] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor. Longest prefix matching using bloom filters. In *SIGCOMM '03*.

[17] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. Routebricks: exploiting parallelism to scale software routers. In *SOSP '09*.

[18] W. Eatherton. The push of network processing to the top of the pyramid. Keynote, ANCS '05.

[19] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron. Energy-efficient mechanisms for managing thread context in throughput processors. In *ISCA '11*.

[20] M. Gebhart, S. W. Keckler, and W. J. Dally. "a compile-time managed multi-level register file hierarchy". In *MICRO '11*.

[21] V. Govindaraju, C.-H. Ho, and K. Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *HPCA '11*.

[22] P. Gupta, S. Lin, and N. Mckeown. Routing lookups in hardware at memory access speeds. In *INFOCOM '98*.

[23] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August. Bundled execution of recurring traces for energy-efficient general purpose processing. In *MICRO '1*.

[24] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *ISCA '10*.

[25] C. Kim, M. Caesar, and J. Rexford. Floodless in SEATTLE: A scalable ethernet architecture for large enterprises. In *SIGCOMM '08*.

[26] 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. In *PACT '10*.

[27] S. Kumar, M. Becchi, P. Crowley, and J. Turner. CAMP: fast and efficient IP lookup architecture. In *ANCS '06*.

[28] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *SIGCOMM '06*.

[29] S. Kumar and B. Lynch. Smart memory for high performance network packet forwarding. In *HotChips*, Aug. 2010.

[30] H. Le and V. Prasanna. Scalable tree-based architectures for ipv4/v6 lookup using prefix partitioning. *IEEE Trans. Comp.*, PP(99):1, '11.

[31] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO 42*.

[32] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. In *SIGCOMM '03*.

[33] D. Taylor, J. Turner, J. Lockwood, T. Sproull, and D. Parlour. Scalable ip lookup for internet routers. *Selected Areas in Communications, IEEE Journal on*, 21(4):522 – 534, may 2003.

[34] D. E. Taylor and J. S. Turner. Classbench: A packet classification benchmark. In *IEEE INFOCOM '05*.

[35] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. Cacti 5.1. Technical Report HPL-2008-20, HP Labs.

[36] M. Thuresson, M. Sjalander, M. Bjork, L. Svensson, P. Larsson-Edefors, and P. Stenstrom. Flexcore: Utilizing exposed datapath control for efficient computing. In *IC-SAMOS '07*.

[37] N. Vaish, T. Kooburat, L. De Carli, K. Sankaralingam, and C. Estan. Experiences in co-designing a packet classification algorithm and a flexible hardware platform. In *ANCS '11*.

[38] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar. Efficuts: optimizing packet classification for memory and throughput. In *SIGCOMM '10*.

[39] G. Venkatesh, J. Sampson, N. Goulding, S. K. V, S. Swanson, and M. Taylor. Qscores: Configurable co-processors to trade dark silicon for energy efficiency in a scalable manner. In *MICRO '11*.

[40] B. Vöcking. How asymmetry helps load balancing. In *IEEE-FOCS '99*.

[41] K. Zheng and B. Liu. V6gene: A scalable IPv6 prefix generator for route lookup algorithm benchmark. In *AINA '06*.

APPENDIX

A. MOTIVATING BACKGROUND WORK

This appendix contains a report prepared for a course project

which formed the foundation and background research for LEAP. The report is presented in its full here:

Specializing PLUG: Performance Improvements to a Tiled Network Processing Framework

Eric Harris Samuel Wasmundt
University of Wisconsin-Madison
{enharris,wasmundt}@wisc.edu

ABSTRACT

Tiled network processing frameworks such as PLUG offer programmability and flexibility in network router hardware. However, such solutions suffer from higher latency, area, and power than single algorithm ASIC circuits. This paper works to replace inefficient general purpose microcores in the PLUG tiled architecture with the bare minimum hardware required. Three algorithms, IPv4 Routing, Ethernet Forwarding, and Ethane routing are implemented as ASIC circuits while retaining the tiled structure and software framework of PLUG. The knowledge gained in designing these circuits is then abstracted into a domain specific language. This domain specific language acts as an intermediate step, allowing for future C++ API to ASIC circuit generation, and helps set the stage for a crossbar based design replacing generalized cores in a tiled network routing architecture. The reconfigurable nature of the crossbar provides further flexibility that is not available with a traditional ASIC design.

1. INTRODUCTION

High-end, network router line cards are a high/throughput, low-latency challenge for computer architects. Increasingly, the power consumption of network equipment is a concern. To meet the demands of this environment, many line cards implement algorithm-specific ASIC designs. ASIC network line cards, however, suffer from high engineering redesign costs every time a new routing algorithm or packet classification structure is introduced.

Programmable solutions such as PLUG[1] return generality to network routing and provide a software framework and toolchain to allow flexibility. Increasingly, software development and verification are becoming the cost-dominant portions of ASIC design [2]. This makes programmable solutions particularly attractive due to faster time to market and significant decreases in NRE costs. However, such solutions are not as efficient as ASIC implementations.

This paper works to leverage the software stack and tiled architecture of PLUG with the latency, power and silicon area gains of ASICs. To do so we replace the micro-cores in the PLUG tiles with the minimum hardware required to complete a given tile's tasks. By analyzing three algorithms (IPv4, Ethernet Forwarding and Ethane[3])

and implementing their tiles via ASIC-style design, knowledge was gained on what computational functionality tiles need to have. These ASIC designed tiles are 100x to 570x more efficient in computational area, reduce total power consumption by 30-40% and reduce latency when compared to the original PLUG implementation. The computational requirements for these algorithms were then abstracted to the domain specific language this paper proposes. This abstraction allows for the design of a multipurpose array of computation units connected with a crossbar proposed in this paper for future research.

2. BACKGROUND

2.1 LINE CARD HARDWARE

Traditionally network line cards are implemented with ASIC designs containing TCAMs. Their algorithm-specific approach requires a hardware redesign, verification and software development with each new backbone router generation. Alternatively, pipelined network processing architectures break up network router tasks such as packet classification and packet routing into a high-throughput pipeline. This allows network routing hardware such as PLUG[1] and others[4] to be programmable. Figure 1 illustrates PLUG's role in network hardware. PLUG overcomes many of the problems with previous programmable network router designs with its tiled architecture shown in figure 1(d). PLUG also has a rich software framework and toolset with a C++ API that routing applications can be developed in. However, PLUG still suffers from higher power, higher latency and a larger silicon footprint than ASIC designs that run a single algorithm.

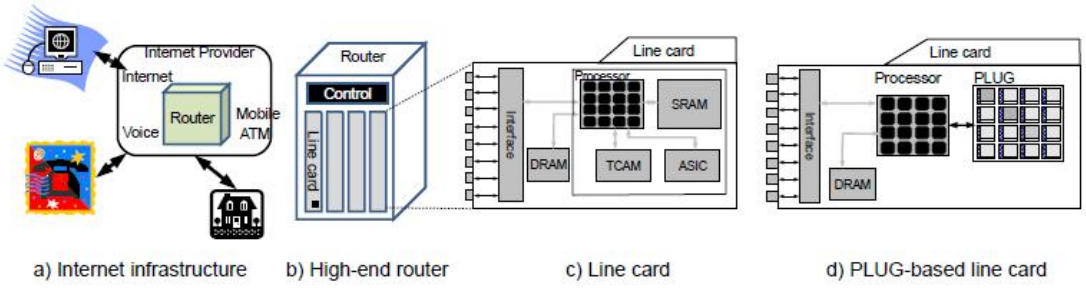


FIGURE 1: PLUG's role in network routing. Figure from [1]

2.2 PLUG ARCHITECTURE

PLUG consists of a grid of tiles for network processing. Each tile contains three things: SRAM data memory, routing logic, and computation circuitry consisting of 32 microcores. The PLUG array can receive up to one message per cycle and each tile must accept a message every cycle. To achieve this high throughput the

computation circuitry is broken up into 32 microcores. A single core gets a new input every 32 cycles and has up to 32 cycles to complete its operation. These 32, 16-bit, cores have a 256 entry instruction memory and 32 registers. The ISA is simple. This paper assumes a 4x4 tile grid, which was previously proven sufficiently large to map many network algorithms.

Algorithms are broken into pipelineable steps called codeblocks. These codeblocks are implemented in the C++ PLUG API. Codeblocks can be thought of as a single function that has at most one memory access. When a PLUG tile gets an input message, the message header contains the id of the codeblock to run. The codeblock can then have at most one memory access and a finite number of computational steps. The codeblock then can output messages to any one of the tile's 6 output networks. PLUG has a compiler and a scheduler that maps routing algorithm implementations onto the physical tile array and generates the appropriate PLUG assembly for each tile.

2.3 SOURCES OF INEFFICIENCY IN THE PLUG ARCHITECTURE

The PLUG tile devotes a significant portion of its area and power to the microcore cluster. Figure 2 shows the approximate area and power breakdown according to estimates from 55nm TSMC Design Compiler synthesis.

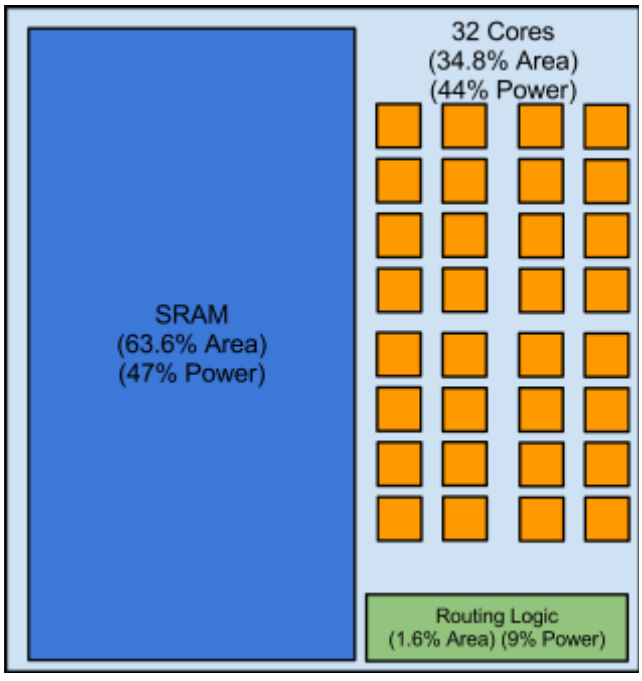


FIGURE 2: Unmodified PLUG Tile Area/Power

Power and area are "wasted" on general purpose elements that are not present in a strictly ASIC implementation of a specific algorithm. For example, an ASIC implementation of IPv4 routing does not have to

contain instruction memories, but rather might have a smaller state machine. Power spent storing values to and from registers in an algorithm running on PLUG is likely not consumed in an ASIC version. Further, power and area inefficiencies come from the inability to express the exact purpose of the algorithm in software. For example, to count the number of 1's in a 32-bit number, the 16-bit PLUG ISA code would need multiple loops with branching and logical AND operations to mask bits. In a strictly hardware implementation this could be implemented efficiently as a bit counter. Similarly, a simple MUX'ing operation in the hardware would have to be represented as a conditional branch in the assembly code running on the general purpose cores.

3. METHODOLOGY

To find the upper bound in performance and efficiency of a given routing algorithm while still retaining the tiled structure of PLUG, specific algorithms were chosen to be implemented with custom hand-designed hardware. This point is not as efficient as a full ASIC design but gives a target for the PLUG hardware to map software algorithms onto. Three programs were chosen for this manual implementation process: Ethernet Forwarding, IPv4 Routing, and Ethane[3]. Ethernet Forwarding simply forwards a packet by looking up in its data structures which port to transmit the packet on. IPv4 has a more complex routing algorithm that requires multiple lookups in order to route a single packet. Ethane is packet routing with a security policy regulating the routing.

3.1 VERILOG DEVELOPMENT

Routing programs were implemented in Verilog by keeping the PLUG tile interface and function constant. The microcores were swapped out of the tile for the ASIC hardware needed. Hardware was pipelined where necessary to avoid the need to duplicate it in order to accept a new input every cycle. Each routing program required different codeblocks to be executed on different tiles, thus specific tiles were developed for each step in the dataflow, not a single general tile that covers the entire algorithm. The tile was assumed to have a customizable, limitless size SRAM. In an actual hardware implementation it is possible that the SRAM storage needed would be broken up into multiple tiles.

Initially, hand drawn schematics were created for each algorithm. This allowed for many efficiencies to surface through this manual implementation process. The process of converting C++ algorithms in Verilog while simultaneously finding optimizations was a challenging but crucial step in the implementation process . Figure 3 below depicts a code sample that was previously compiled down into a long sequence of sequential instructions on

the PLUG ISA. The purpose of the code is to count the number of bits up to the variable count. This can be implemented in hardware with a mask generator, a logical AND, and a bit counter. Similar instances, where the custom hardware can efficiently complete the software's task without a long sequence of instructions, are found throughout the three programs implemented.

```
for(i=0; i<=count; i++){
   if(i<PLUG_MSG_SIZE){
    if(((bitmap_hi<<i)&0x8000) == 0x8000)
    {
      if(index_lo == 0xFFFF){
        ++index_hi;
        index_lo = 0x0000;
      }
      else ++index_lo;
    }
   }
   else{
    if(((bitmap_lo<<(i-PLUG_MSG_SIZE))&0x8000) == 0x8000)
    {
      if(index_lo == 0xFFFF){
        ++index_hi;
        index_lo = 0x0000;
      }
      else ++index_lo;
    }
   }
  }
```

FIGURE 3: Example Obfuscated Code

3.2 VERILOG VERIFICATION AND TESTING

In order to verify the custom hardware implementation a testing framework was developed. For ease and speed of testing and debugging the hardware implementation, which replicated the functionality of the original PLUG hardware, the algorithms were verified only on a per-tile basis. Algorithms written in the PLUG API are already verified on a routing program level[1], thus we use the argument that verification on a per tile basis is sufficient to verify program correctness because no other modifications were introduced to the existing framework.

The PLUG toolset comes with a tile-level simulator. However, at the time of this paper it was unable to generate per-tile traces for all of the routing programs implemented. To generate per tile traces, the C++ PLUG API implementation of the program was annotated with code to generate trace files at every tile message input and output message, see Figure 4 below for an example. The same program input set used to test the C++

implementation was used to generate the traces. Test inputs for the implemented algorithms included thousands of test look-ups. These look-ups are the time critical function as opposed to the read and update operations which are less time sensitive for backbone router applications.

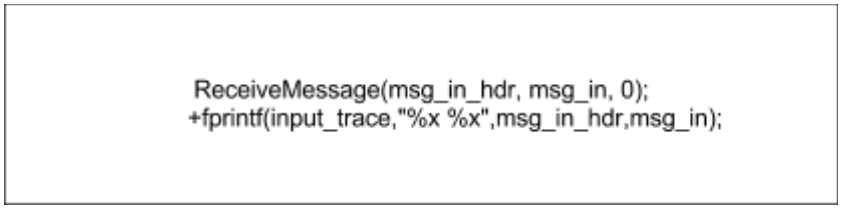


FIGURE 4: Annotated C++ Program Implementation Example

After the input and output traces were generated they were fed into a self-checking Verilog testbench. With IPv4 the SRAMs were initialized with a sample routing dataset also used on the C++ implementation verification.

3.3 SYNTHESIS AND ANALYSIS METHODOLOGY

Verilog designs were synthesized in isolation from the tile's SRAM and routing logic, which were not touched in this work. They were synthesised by Design Compiler B-2008.09-SP3 using the same TSMC 55nm standard cell library and wireload model as the original PLUG paper so comparisons were accurate. As with the original PLUG design, synthesis was done at twice the target clock frequency of 1Ghz to get synthesized results with positive slack with the target clock frequency. Dynamic power estimates, total area estimates and timing estimates are all taken directly from the output of Design Compiler and compared to the original PLUG tile. Post synthesis simulation was completed on some tiles to validate synthesis results.

For latency calculations a 4x4 PLUG grid was assumed. Propagation delays of 1 clock cycle (1 ns) were added for data to travel between tiles. Additionally, since in PLUG the output comes out of a fixed location at the lower left corner of the grid, latencies were extended to include the delay to get the output value to that location. This methodology mirrors that of the PLUG paper[1] to keep comparisons fair.

3.4 CHALLENGES

This project required an understanding of and modification to an existing framework. Finding optimizations and extracting the optimal implementation of a given task in software required skill, especially since the software was represented and constrained by the limitation of sequential 16bit PLUG API code.

One specific impediment encountered that should be documented is the flawed design of the bit counter proposed by [5]. Figure 4 in [5] (Figure 5 in this paper) depicts an optimization to an 8 bit bit-counter. However, the carry and sum bit lines are labeled wrong in multiple places so that the circuit the figure depicts does not properly count bits. For example, the input 0x80 should produce the answer 0x1 but produces 0x2.

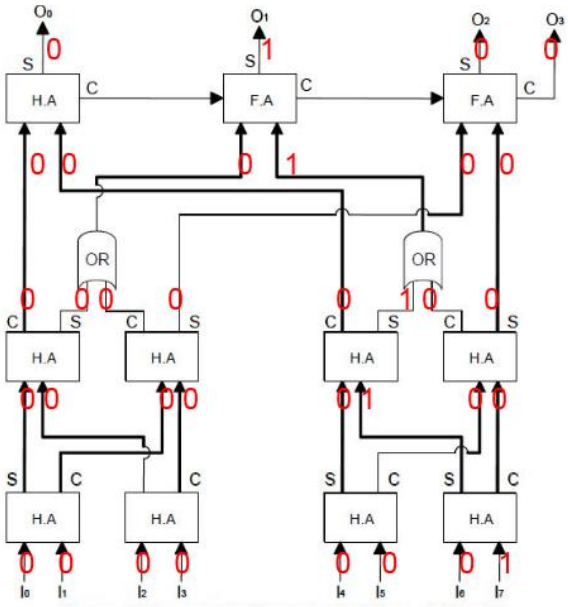


FIGURE 5: Faulty Bit-Counter Implementation with annotated invalid output

4. RESULTS

Three algorithms, IPv4, Ethernet Forwarding, and Ethane were successfully implemented in Verilog and verified. Though not every modern network processing algorithm was implemented into Verilog by hand, patterns developed out of the three that were implemented. Patterns within algorithms emerged such as a need for common functional units across codeblocks within a tile. Additionally, we were able to implement multiple routing protocols with a similar design that hinted at ease of future automation. Significant gains were observed in area and latency, as well as improvement in dynamic power.

4.1 AREA

With the original PLUG implementation, every tile contained 32 microcores, whether or not they were fully utilized by the network application being run. Table 1 summarizes the area of the tiles implemented. Our results showed a 100x to 570x improvement in area devoted to computation, making it a minor portion of total tile area as can be seen in Figure 6 (b) compared to the original in Figure 6(a).

TABLE 1: Area Improvements to PLUG Tile

Benchmark	Tile	Tile Area (um^2)	Computational Area (um^2)	Percent of Tile Area for Computation
Original PLUG	*PLUG Tile*	*3,246,045.57*	*1,130,397.44*	34.82%
IPv4	L1	2,120,801.33	5,153.20	0.24%
IPv4	L1 Rule	2,119,289.93	3,641.80	0.17%
IPv4	L1 Child	2,119,552.73	3,904.60	0.18%
IPv4	L2	2,124,886.73	9,238.60	0.43%
IPv4	L2 Rule	2,120,896.33	5,248.20	0.25%
IPv4	L2 Child	2,119,630.33	3,982.20	0.19%
IPv4	L3	2,121,200.93	5,552.80	0.26%
IPv4	L3 Rule	2,118,341.13	2,693.00	0.13%
Ethernet Fwd	head_core	2,119,171.13	3,523.00	0.17%
Ethernet Fwd	core	2,117,311.53	1,663.40	0.08%
Ethane	dleft_flow	2,120,134.10	4,485.97	0.21%
Ethane	flow	2,120,550.91	4,902.78	0.23%
Ethane	forbidden	2,119,215.71	3,567.57	0.17%
Ethane	head_forbidden	2,121,039.70	5,391.57	0.25%
Ethane	port	2,118,719.71	3,071.58	0.14%

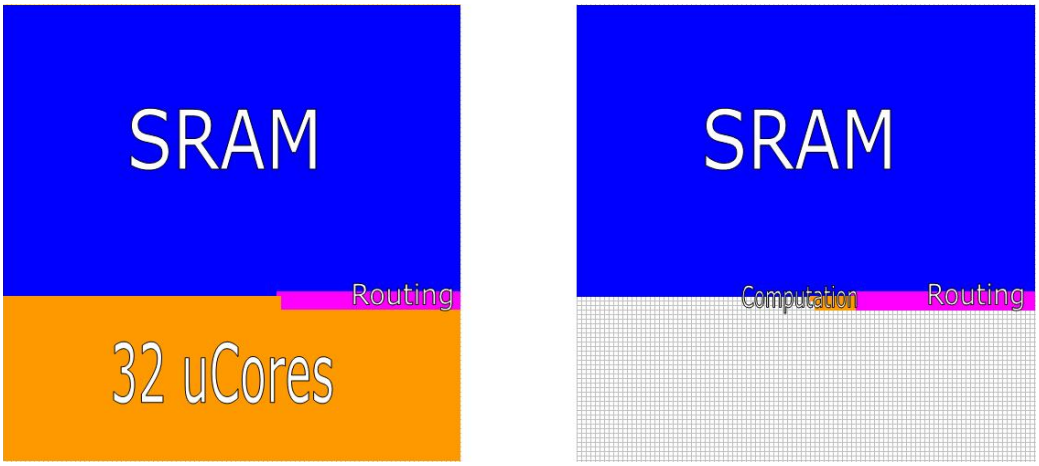


FIGURE 6: (a) Left: Original PLUG Tile Proportional in Area (b) Right: L2 Tile Proportional in Area

4.2 POWER

Similarly, the dynamic power was reduced for every tile implemented when compared to the original PLUG tile. Table 2 and Figure 7 show the power results. Unlike the area, computation still consumes a non-trivial amount of tile power, but total tile power is reduced by 30% to 41% when comparing to the original PLUG tile.

TABLE 2: Dynamic Power Improvements to PLUG Tile

Benchmark	Tile	Tile Power (mW)	Computational Power (mW)	Percent of Tile Power for Computation	Percent Original PLUG Tile Power
Original PLUG	*PLUG Tile*	*48.27*	*21.28*	44.08%	100.00%
IPv4	L1	31.83	4.8377	15.20%	65.94%
IPv4	L1 Rule	31.18	4.1934	13.45%	64.60%
IPv4	L1 Child	31.28	4.284	13.70%	64.79%
IPv4	L2	33.82	6.8267	20.19%	70.06%
IPv4	L2 Rule	31.76	4.7702	15.02%	65.80%
IPv4	L2 Child	31.32	4.3335	13.83%	64.89%
IPv4	L3	32.05	5.0612	15.79%	66.40%
IPv4	L3 Rule	29.86	2.8648	9.60%	61.85%
Ethernet Fwd	head_core	30.47	3.4777	11.41%	63.12%
Ethernet Fwd	core	28.35	1.3621	4.80%	58.74%
Ethane	dleft_flow	29.96	2.9734	9.92%	62.08%
Ethane	flow	31.75	4.7632	15.00%	65.78%
Ethane	forbidden	28.34	1.3478	4.76%	58.71%
Ethane	head_forbidden	31.71	4.7183	14.88%	65.69%
Ethane	port	29.51	2.5224	8.55%	61.14%

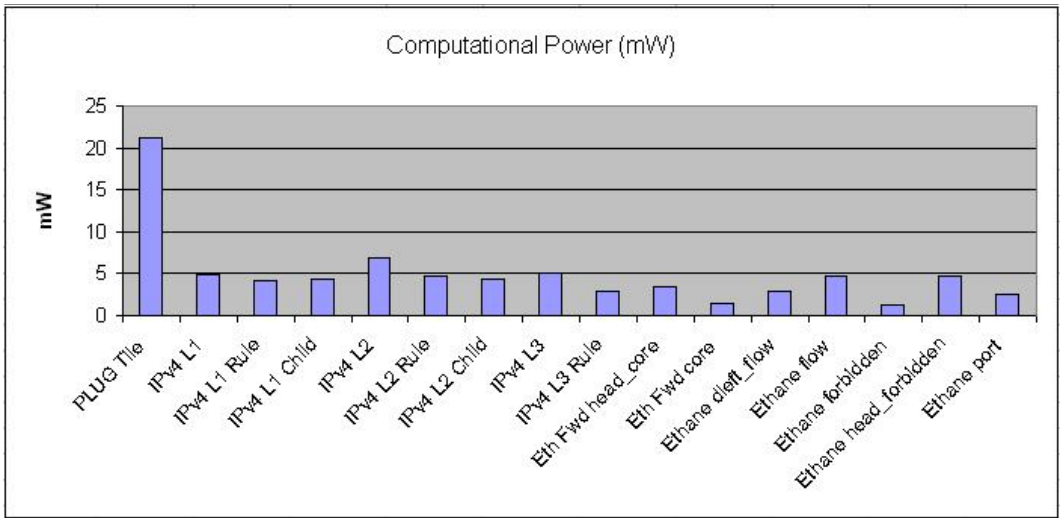


FIGURE 7: Computational Power Improvement

4.3 LATENCY, THROUGHPUT

All our tiles met timing at 1 GHz which matches the cycle time of the original PLUG implementation. Since we retained the same tile interface and structure and did not modify the algorithms' dataflow paths between the tiles, throughput stayed constant at one message per cycle. However, our implementation reduces the latency because loops and branching that took multiple cycles in the original PLUG design do not surface

in our implementations. In Ethernet Forwarding and Ethane, the latency is actually less than reported, but extra communication delay is used to propagate the response to the output of the 4x4 tile grid. Figure 8 summarizes the latency improvements for the three implemented network applications.

Program	PLUG Latency	New Latency
IPv4	92	24
Ethernet Fwd	20	11
Ethane	41	13

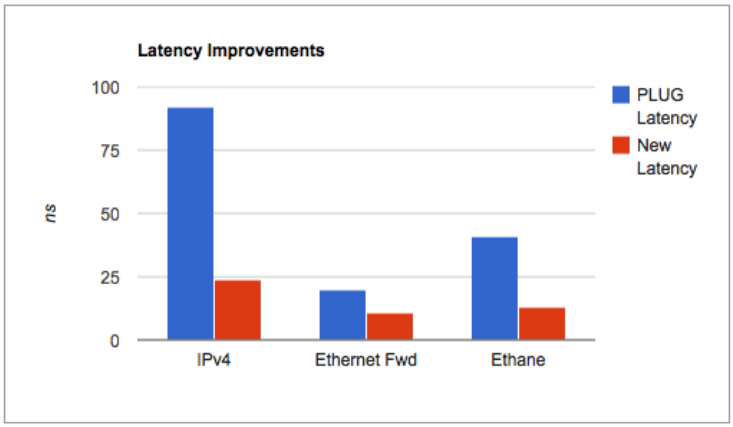


TABLE 3 & FIGURE 8: Latency Improvement

5. DOMAIN SPECIFIC LANGUAGE

In order to return a little generality to the specific Verilog tile implementations, a domain specific language was created for PLUG. This domain specific language was designed so that the Verilog implementation of a program coded in it would be easily derived. This domain specific language has common operations for use in multiple tiles that were identified from the hand generated hardware implementations. By writing a codeblock in this implementation, the data flow graph should be fairly easy to extract, programatically or by inspection. This allows either a tool or a hardware designer to see potential areas to pipeline the hardware. See Figure 9 for an example of this domain specific language and a corresponding data-flow graph. The domain specific language also helps identify which elements different codeblocks on the same tile, and different tiles have in common. This could be used to help generate a generic tile that covered the computational requirements of all the codeblocks in a network program. Thus, with this process, hardware could be created that covers many routing algorithms with some programability and generality that a strict ASIC design could not have. Such a design still could be more power, latency and area efficient than the original PLUG design. The domain specific language was also designed to be easy to automatically generate, Section 7 discusses future plans for the process of converting to and from the domain specific language and how it will be automated in the future.

```
RdMem(InMsg[0][15:5],ALWAYS)
MASKED_BITCOUNT[0]{32}(InMsg[0][4:0],{data[31:16],data[47:32]})
MASKED_BITCOUNT[1]{32}(InMsg[0][4:0],{data[79:64],data[95:80]})
ADD[0]{16}(MASKED_BITCOUNT[0],data[15:0]) #R_Index Incremented
ADD[1]{16}(MASKED_BITCOUNT[1],data[63:48]) #C_Index
DECREMENT[0]{16}(ADD[0]) #R_Index
DECREMENT[1]{16}(ADD[1]) #C_Index
OutMsg[0] = {DECREMENT[1],DECREMENT[0]}
Send(0,ALWAYS)
```

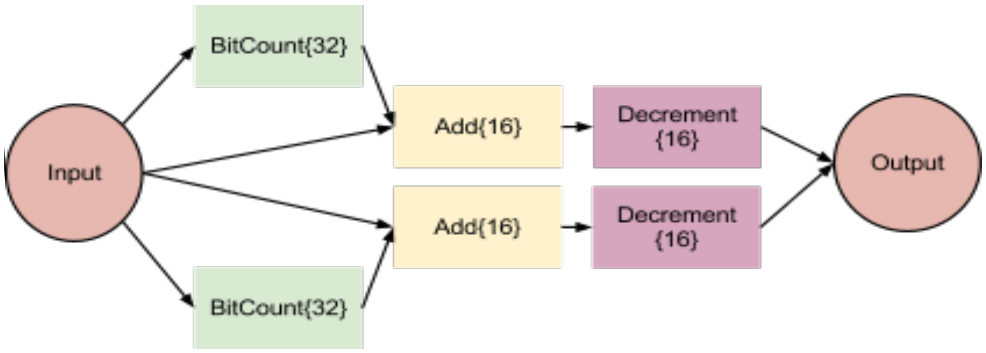


FIGURE 9: Domain Specific Language Sample with Dataflow Diagram

5.1 MINIMUM COMPUTATIONAL REQUIREMENTS

From the domain specific language, the required computational units for a given protocol for a given tile can be extracted. Table 4 summarizes the requirements for the implemented tiles. This table is a result of taking the arbitrary width operations of the algorithms in the domain specific language and attempting to map them to 6 types of compute elements: 16-bit adders, 32bit bit counters, 16 bit logical operations, 16 to 1 Mux (8 bits wide), 2 to 1 Mux (16 bits wide) and a binary search tree with a comparison at each node of 8 bits. Logic shared across all tiles such as hardware to extract the codeblock number from the message header is not included in the table. (Note: These numbers may not be the optimal mapping but serve as a good indicator of the computational requirements).

Program	Core	16 bit adders	32bit Counters	16 Logical Op	16MUX1	2MUX1 16	3Deep 8 Bit BST
IPv4	L1	4	2	0	1	5	0
IPv4	L1_Rule	0	0	1	0	5	0
IPv4	L1_Child	1	0	0	0	6	0
IPv4	L2	6	2	2	1	5	1
IPv4	L2_Rule	0	0	1	1	6	0
IPv4	L2_Child	1	0	0	0	2	0
IPv4	L3	2	1	1	0	5	1
IPv4	L3_Rule	0	0	1	0	1	0
EthernetForwarding	Head	1	0	3	0	0	0
EthernetForwarding	Normal	0	0	3	0	0	0
Ethane	dleft	0	0	6	0	3	0
Ethane	flow	0	0	6	0	3	0
Ethane	forbidden	0	0	7	0	5	0
Ethane	head_fort	0	0	7	0	5	0
Ethane	port	0	0	0	0	0	0

TABLE 4: Tile Requirements Estimate

From this table, a generalized tile hardware that could cover multiple tiles' computational requirements could be created. This would restore the generality of the PLUG Tile and allow many (but not all) routing algorithms to be mapped to this general tile. One thing to note is that no tile needed the 32 adders that were originally in the PLUG tile, one in each core. It is a similar story for the other elements, where the routing algorithms did not fully utilize the 32 cores' capabilities, especially when they were looping to do a simple operation like bit counting.

6. CROSSBAR

Work was also started to implement the PLUG version of Efficuts[6] in the domain specific language. Efficuts differs from other network programs in that it is a packet classification application and requires a relaxing some of the constraints of the original PLUG including single memory accesses per codeblock, maximum codeblock lengths and throughput requirements. With some of Efficuts and the three standard routing algorithms in the domain specific language, a natural abstraction of the computation hardware arose.

The computation required in each tile can be thought of as a series of steps in a dataflow diagram. Each step requires the use of one or more available computational blocks and each computational block receives data from either the inputs to the tile, the SRAM or another computational block. Similarly each compute unit outputs its result to other computational units or the output. If the computational units required for an algorithm are linked with a crossbar, then the data operations can be sequenced. Figure 10 demonstrates this concept.

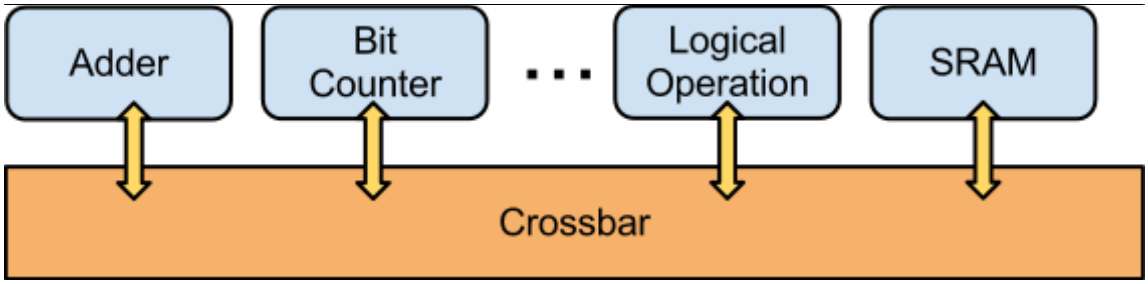


FIGURE 10: Crossbar Abstraction

7. FUTURE WORK

This work was the first portion of a larger project to optimize the PLUG Framework. Future work will automate the domain specific language generation from routing programs written in the PLUG C++ API by using the LLVM back end to compile down to the domain specific language. Future work will also be able to generate Verilog automatically from the domain specific language. This, in isolation, creates a chip generator that works to "codify designer knowledge" as suggested by [7]. In addition to the 3 algorithms that we already have implemented and our work on Efficuts, we plan on implementing IPv6 and all other algorithms that have been implemented on the general purpose microcores of the PLUG architecture to show that our implementation process can handle everything that was previously supported.

Future work will also create hardware for the crossbar abstraction discussed in section 6. With a working crossbar implementation, PLUG can retain most of its generality and programmability while taking advantage of some of the benefits of ASIC designs.

8. CONCLUSION

We have shown that removing the microcores in the PLUG tile can achieve the low area, latency and power gains of an ASIC style design while retaining some of the flexibility and tools offered by the PLUG framework. Our optimizations showed 100x to 570x computational area improvement, 30%-40% decrease in total dynamic power consumption, and a significant reduction in latency. Through our design process we were able to gain valuable insight into similarities among code blocks and across algorithms. To represent some of these insights and optimizations, a domain specific language was developed. This domain specific language allows easy extraction of

the dataflow graphs for a given algorithm and is designed to be easy for use with automated hardware generators. By developing algorithms in this language, the groundwork has been laid for further research into a crossbar style implementation for PLUG that can cover most algorithms written in the PLUG API.

7.ACKNOWLEDGEMENTS

This project was completed under the supervision, insight and feedback from UW-Madison Professor Karthikeyan (Karu) Sankaralingam We would like to thank UW-Madison graduate students Lorenzo De Carli and Jin Kim for helping us understand and use the existing PLUG framework. UW-Madison graduate student Chenhan Ho also assisted with standard cell synthesis. We would also like to thank the Vertical Research Group at UW-Madison for their early input on the project. A special thanks to Mark Hill for teaching his CS 752 architecture course.

8. REFERENCES

[1] A. Kumar et al., "Design and implementation of the PLUG architecture for programmable and efficient network lookups," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques (PACT)*, New York, NY, USA, 2010, pp. 331–342.

[2] R. Avinun, "Validate hardware/software for nextgen mobile/consumer apps using software-on-chip system development tools," *TechOnline India*, 05-May-2011. <http://www.techonlineindia.com/article/11-05-05/Validate_hardware_software_for_nextgen_mobile_consumer_apps_using_software-on-chip_system_development_tools.aspx>

[3] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: taking control of the enterprise," *in Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, New York, NY, USA, 2007, pp. 1–12.

[4] T. Sherwood, G. Varghese, and B. Calder, "A pipelined memory architecture for high throughput network processors," in *30th Annual International Symposium on Computer Architecture (ISCA), 2003. Proceedings*, 2003, pp. 288- 299.

[5] Dalalah, Ahmed & Sami Baba. "New Hardware Architecture for Bit-Counting." 5*th WSEAS International Conference on Applied Computer Science*. Hangzhou, China. April, 2006. pp.118-128

[6] Vaish, N.;,Kooburat, T.;,De Carli, L.;,Sankaralingam, K.;,Estan, C.; , "Experiences in Co-designing a Packet Classification Algorithm and a Flexible Hardware Platform," *Architectures for Networking and Communications Systems (ANCS), 2011 Seventh ACM/IEEE Symposium on* , vol., no., pp.189-199, 3-4 Oct. 2011

[7] O. Shacham, O. Azizi, M. Wachs, S. Richardson, and M. Horowitz, “Rethinking Digital Design: Why Design Must Change,” *IEEE Micro*, vol. 30, no. 6, pp. 9-24, Nov. 2010.