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

Eric N. Harris† Samuel L. Wasmundt† Lorenzo De Carli†
Karthikeyan Sankaralingam† Cristian Estan‡
†University of Wisconsin-Madison ‡Broadcom Corporation
enharris@uwalumni.com {wasmundt,lorenzo,karu}@cs.wisc.edu cestan@broadcom.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.

We build on recent innovations in computer architecture that demonstrate the efficiency and flexibility of dynamically synthesized accelerators. In this paper we propose LEAP, a latency- energy- and area- optimized lookup pipeline based on an analysis of various lookup algorithms. We compare to PLUG, which relies on von-Neumann-style programmable processing. We show that LEAP has equivalent flexibility by porting all lookup algorithms previously shown to work with PLUG. At the same time, LEAP reduces chip area by $1.5\times$, power consumption by $1.3\times$, and latency typically by $5\times$. Furthermore, programming LEAP is straight-forward; we demonstrate an intuitive Python-based API.

Categories and Subject Descriptors

B.4.1 [**Data Communication Devices**]: Processors; C.1 [**Computer Systems Organization**]: Processor Architectures

Keywords

Network processing, Lookups, TCAM, Dynamically-specialized datapath

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. Software based table lookups [17], lookup hardware integrated into the packet processing chip [18], and dedicated lookup chips [3, 4] are different implementations with the latter two being the preferred industry approach.

We observe that there is an increasing sophistication in the lookup processing required and reducing benefits 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 paper's goal is to investigate a new class of flexible lookup engines with reduced latency, energy consumption, and silicon area that 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 architecture we propose is called LEAP. It is meant to act as a co-processor or lookup module for network processors or switches, not to implement the complete data plane processing for a packet. 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, we use a dynamically configurable data path. We analyze seven algorithms for forwarding lookups and packet classification to determine a mix of functional units suitable for performing the required processing steps. We

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ANCS'12, October 29–30, 2012, Austin, Texas, USA.
Copyright 2012 ACM 978-1-4503-1685-9/12/10 ...$15.00.

have designed, implemented in RTL, and verified one instance of the LEAP architecture 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 $5\times$.

The remainder of this paper is organized as follows. We first present motivating background and related work in Section 2. Section 3 presents a characterization of lookup processing, Section 4 presents the LEAP architecture, Section 5 discusses programming mechanisms and our Python API. Section 6 presents quantitative evaluation of LEAP based on a diverse set of seven lookup algorithms. Section 7 concludes the paper.

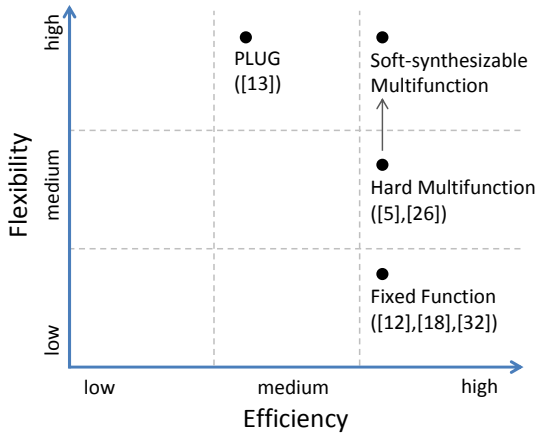


Figure 1: Design space of Lookup Engines

2. BACKGROUND AND MOTIVATION

We first describe background to place our work in context. Two main approaches exist for implementing lookups, namely, run-to-completion (RTC) architectures and dedicated lookup-based engines. The RTC paradigm exploits packet-level parallelism: the core in charge of each packet fully performs the lookup based on the data in the packet. Lookup data structures are maintained by the network processor on a globally shared memory (across the processor's cores). 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), relevant fields from packet headers are moved close to relevant lookup data structures. The goal is to minimize the amount of data moved around. In principle, both approaches are equally valid. In this work we focus 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.

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 [20, 31, 37, 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. 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 and we elaborate on this design space below.

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 [32], 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.

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 [26]. Baboescu *et al.*[5] propose a circular pipeline that supports IPv4 lookup, VPN forwarding and packet classification. Huawei is building tiled Smart Memories [28] that support 16 functions including IP forwarding lookup, Bloom filters that can be used in lookups [16, 15], sets and non-lookup functions 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 [37].

While PLUG attains high flexibility, it lacks high efficiency. It is instructive to understand why it has inefficiencies. Using the McPAT [30] modeling tool, we modeled a simple 32-bit in-order core similar to a PLUG microcore. In Figure 2, we show the percentage contribution of energy consumption of this processor due to its four main primitive functions: instruction-fetch and decode, register-file read, execute, and write-back (includes register write, pipeline staging etc.). Events other than execute are *overhead* yet they account for more than 80% of the energy. While it may seem surprising,

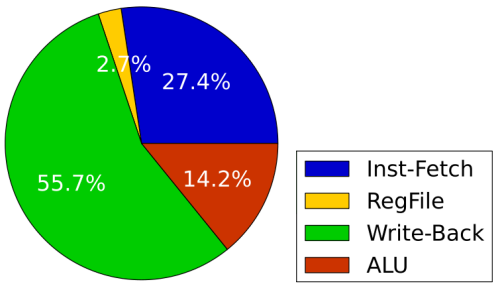


Figure 2: Programmable processor energy breakdown.

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. We are not the first to make this observation. In the realm of general purpose processors, Hameed *et al.*[22] 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. 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. We draw 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 [19], FlexCore [35], QsCores [38], and BERET [21]. 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, we leverage 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

We now characterize the computational requirements of lookups. We assume a dataflow approach where processing is broken up in steps and mapped to a tile based pipeline similar to PLUG. We focus on the processing to understand how to improve on existing flexible lookup approaches and incorporate the insights of hardware specialization.

3.1 Description of Lookup Algorithms

We examined in detail seven lookup algorithms. 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. Much of our analysis was in understanding the basic steps of the lookup processing and determining the hardware requirements by developing design sketches as shown in Figure 3. We describe this analysis in detail for two algorithms, Ethernet forwarding and IPv6 and the next section summarizes overall findings. These two serve as running examples through this paper, with their detailed architecture discussed in Section 4.4 and programming implementation discussed in Section 5.3.

Context	Approach/Key data structures
Ethernet forwarding	D-left hash table [8, 10, 39]
IPv4 forwarding	Compressed multi-bit tries [14]
IPv6 forwarding	Compressed multi-bit tries + hash tables [25]
Packet classification	Efficuts with parallel decision trees [37, 36]
DFA lookup	DFA lookup in compressed transition tables[27]
Ethane	Parallel lookup in two distinct hash tables [11]
SEATTLE	Hash table for cached destinations, B-tree for DHT lookup [24]

Table 1: Characterization of lookup algorithms.

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. Our 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 two keys match. The input is also forwarded to other tiles – this enables multiple tiles to check all the entries in a bucket in parallel.

IPv6 lookup

Application overview: The IPv6 forwarding approach discussed here is derived from PLUG, and is originally based on the "Lulea" algorithm [14]. This algorithm uses compressed multibit tries with fixed stride sizes. The IPv6 lookup algorithms extends it by using two pipelines operating in parallel. Pipeline #1 covers prefixes with lengths up to 48 bits, and is implemented as a conventional multi-bit trie with five fixed-stride stages. Pipeline #2 covers prefixes of length between 48 and 128 bits, and is implemented as a combination of hash tables and trie. If both pipelines return a result, pipeline #2 overrides #1.

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. In the following we describe one of the processing steps for computing a rule pointer in IPv6.

IPv6 rule lookup step: This step, represented in Figure 3b, uses a subset of the IPv6 address bits to construct a pointer into a forwarding rule table. Specifically, it deals with the case where the 256

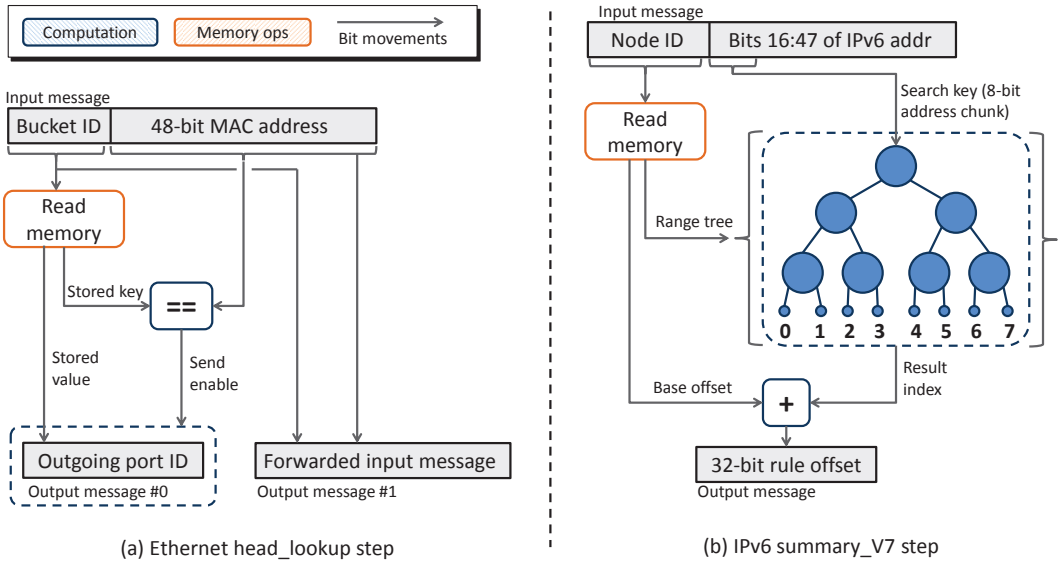


Figure 3: Rule offset computation in IPv6

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 IPv6 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 IPv6 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, we 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. We enumerate these below, tying back to our discussion in Section 2.3 and conclude with the elements of an abstract design.

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, one `bstsearch` instruction that does a binary search through 16-bit chunks of a 128-bit value, like used in [29], can replace a sequence of `cmp, shift` instructions. There is great 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.

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 us 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. In the next section, we describe the LEAP architecture, which is an implementable realization of this abstract design.

4. LEAP ARCHITECTURE

In this section we describe the architecture and implementation of the LEAP lookup engine. First, we present its organization and execution model and discuss its detailed design. We then walk

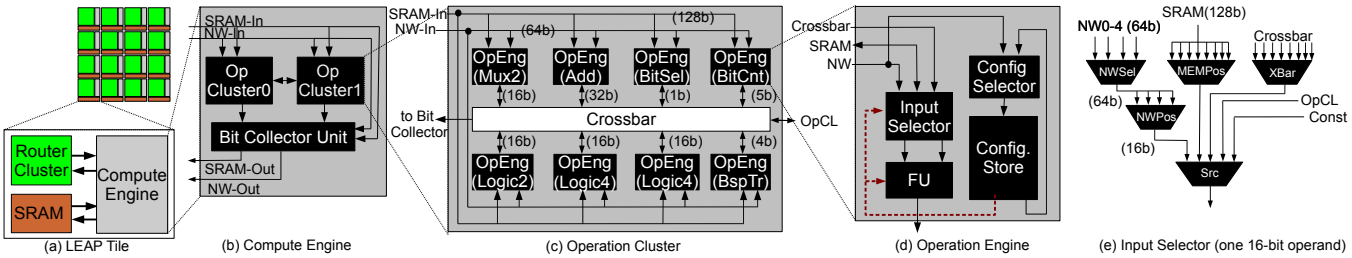


Figure 4: LEAP Organization and Detailed Architecture

through an example of how lookup steps map to LEAP. We conclude with a discussion of physical implementation and design tradeoffs. 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. In this paper, we discuss in detail a mesh-based chip organization and integration with SRAM.

4.1 Hardware Organization

For clarity we first present LEAP assuming all computation steps are one cycle. Section 4.5 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. We reuse the same tiled design as PLUG ([25]) 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. We first summarize the chip-level organization before describing the details of LEAP. At the chip-level, the router cluster in each tile is used to a form a mesh network across the tiles. We mirror 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. PLUG used six inter-tile networks, but our analysis showed only four were needed. Each network is 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, we can process lookups that are up to 256 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, we allow the following seven types of primitive hardware operations 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. In our design, a single Compute Engine was sufficient. With this execution model, the architecture sustains a throughput of one lookup every cycle. As described in Section 4.6, our prototype runs at 1 GHz, thus providing a throughput of 1 billion lookups per second.

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

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 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. We support 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 (but this computation must not conflict with any configuration of computation in the C stage). Our analysis showed this sufficient.

For packet classification, the SRAM was *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.

Compilation: Lookup processing steps can be specified in a high-level language to program the LEAP architecture. Specifically, we have developed a Python API and used it to implement several lookups (details in Section 5). 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.

4.2 Design

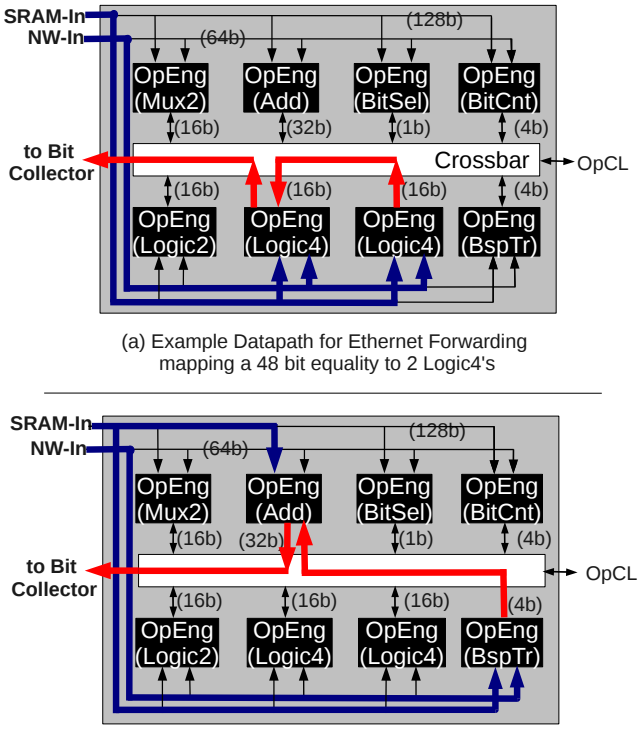
Tile (Figure 4(a)): A single tile consists of one LEAP compute-engine, interfaced to the router-cluster and SRAM.

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. Our 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, we build 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 5. 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. We provide 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. In LEAP, we effectively have 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, but most 16-bit operands each require: `Src` (3 bits), `NWPos` (2 bits), `NWSel` (2 bits), `MEMPos` (3 bits), `XBar` (3 bits), and a `Const` (16 bit). These configuration bits correspond to the input selection shown in Figure 4(e). Section 5 provides a detailed example showing the Python-API and its compiler-generated configuration information.



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

(b) Example Datapath for IPv6

Figure 5: Dynamically created datapaths.

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 operation engine. The compiler is aware of the timing of each operation engine and only chains operation engines together in paths that fit within the single cycle compute-step. 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 Implementation

Based on workload analysis, we arrived at the mix of functional units and the high-level LEAP design. We have completed a prototype implementation of the compute engine in Verilog. We have also synthesized the design along with the associated SRAM mem-

ory to a 55nm technology library using the Synopsys design compiler. Since the compute engine occupies a very small area, we use high-performance transistors which provide high-frequency and low-latency operations - their leakage power is not a large concern. Using these synthesis results we obtain the energy consumed by a lookup access, which we then use to compute power. For SRAM, we consider memories built with low-standby power transistors. The partitioned design restricts a single operation cluster to eight operations and meets timing with a clock period of 1ns. The SRAMs dictate final frequency.

In Section 2.3 we argued that instruction fetch, decode and register files incur area and energy overheads. LEAP eliminates many of these overhead structures. Area and energy costs are now dominated by computation. Detailed quantitative results are in Section 6.

4.4 Mapping lookups to LEAP's architecture

To demonstrate how lookup steps map to LEAP, we revisit the examples introduced in Section 3.1. Figure 5 shows how the steps shown in Figure 3 are configured to run on LEAP.

In our 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 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 5 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 IPv6 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.

4.5 Multi-cycle compute step

LEAP can easily be extended to handle sophisticated lookups requiring multiple *C* stages.The functional units are augmented with a data-store that allows buffering values between compute steps. In the interest of space and clarity we defer a more detailed description to [23].

4.6 Discussion of Tradeoffs

Functional unit mix: Based on our analysis from Section 3.1, we implemented in Verilog specialized hardware designs to determine an appropriate functional-unit mix(details in Section 6). We found that various lookups use a different mix of a core set of operations, justifying a dynamically synthesized lookup engine. Table 3 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].

Algorithm	Critical Path
Ethernet Forwarding	Logic4→Logic4
	Add
IPv4 forwarding	BitCnt→Add→Mux2
	BRPTr→Mux2→Add
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

Table 3: Examples of processing steps' critical paths.

Crossbar design: Instead of designing a "homogenous" cross-bar 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 6. With more area, more operation engines can be integrated into a compute engine. However, this will increase the latency 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.

5. PROGRAMMING LEAP

We now describe the programmer's abstract machine model view of LEAP, a specific Python API we have implemented, and outline an example in detail showing final translation to LEAP configuration bits. The API provides a familiar model to programmers despite our unique microarchitecture.

5.1 Abstract machine model

The abstract machine model of LEAP hides the underlying concurrency in the hardware. Specifically, the programmer assumes a

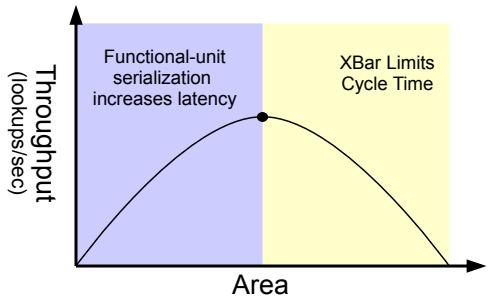


Figure 6: Design scalability tradeoff

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.

5.2 Python API for LEAP

We have developed a simple Python API to make programming LEAP practical. In our model, programmers express their computational steps as Python sub-routines using this API. We chose Python because it is simple to import our API as a module, but we do not rely on any Python-only functionality. Alternatively we could have chosen to implement our API in a non-interpreted language such as C.

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, our 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 4 describes the most important calls in our 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.

```
 1 # In Msg:   0:15: Previous rule
 2 #          16:31: Pointer to target node
 3 #          32:63: Bytes 16:47 of IPv6 address
 4 # Out Msg0: 0:23: Bits 24:47 of IPv6 address
 5 #          24:39: Pointer to new rule
 6 #          40:55: Previous rule
 7 # Out Msg1: 0:31: Pointer to child (unused)
 8 def cb0(nw, nw_out):
 9   data = read_mem(nw[0].body[16:31])
10   ret = bspsearch3_short(nw[0].body[32:39],
11           data[32:87],data[88:95],0x053977)
12   offset = add(data[0:31], ret)
13   nw_out[0].head[8:17] = offset[2:11]
14   nw_out[0].body[0:23] = nw[0].body[40:63]
15   nw_out[0].body[24:39] = offset[16:31]
16   nw_out[0].body[40:55] = nw[0].body[0:15]
17   send(nw_out[0], 1)
18   nw_out[1].head[8:17] = offset[2:11]
19   nw_out[1].body[0:31] = 0xFFFFFFFF
20   send(nw_out[1], 1)
```

(a) Python code

Unit	ADD (a)	ADD (b)	BSPTr (val)	BSPTr (vec)	BSPTr (cfg)	BSPTr (res)	RAM RD	RAM WR
Src	001 (MEM)	002 (XBar)	000 (NW)	001 (MEM)	001 (MEM)	003 (CFG)	00 (NW)	X
NWPos	X	X	11	X	X	X	1	
NWSel	X	X	00	X	X	X	00	X
Mask								X
MemPos	00	X	X	0x4	0xB	X		
Const	X	X	X	X	X	0x53977	X	
XbarPos			X		X			
Xbar	X	111 (BSPTr)	X	X	X	X	X	X
OpCl	X	X	X	X	X	X	X	X

(b) LEAP configuration bits

Figure 7: IPv6 example compute step

5.3 Implementing Lookups in LEAP

To give a sense practical application development, we briefly discuss the implementation of the IPv6 processing step introduced in Section 3.1.

As previously discussed, this compute step works by searching a binary tree using 8 bits of the IPv6 address as the key. The result represents the relative offset of a forwarding rule corresponding to the address. The absolute offset is then obtained by adding the result of the search to a base offset. Both the binary tree and the base offset are obtained by retrieving a 96-bit word from memory, consisting of:

1) Bits 0-31: Base rule offset
2) Bits 32-87: Binary tree, stored as 7-entry vector
3) Bits 88-95: Vector bitmask (specifies valid entries)

Figure 7(a) depicts the LEAP implementation of this step, together with input and output messages. The node is retrieved from memory in line #9. Lines #10-11 perform the search in the range vector. In line #12 the result of the search (index of the range where the prefix lie) is added to the base rule offset to generate a rule index for the next stage. The rest of the code prepares outgoing messages.

The API calls in Figure 7(a) are translated to the configuration bits shown in Figure 7(b). The rows in Figure 7(b) correspond to the generalized input selection shown in Figure 4(f). X's are "don't cares" in the figure and a grayed box indicates those config bits are not present for the particular operand. Each operand in the table has different available config bits because of the different widths and possible sources for each operand.

6. EVALUATION

We now present our evaluation. After describing the methodology for our evaluations, we discuss characteristics of the lookups

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 4: Python API for programming LEAP

Algorithm	Lookup data
Ethernet forwarding	Set of 100K random addresses
IPv4 forwarding	280K-prefix routing table
IPv6 forwarding	Synthetic routing table [40]
Packet classification	Classbench generated classifiers [33]
DFA lookup	Signature set from Cisco [2]
Ethane	Synthetic data based on specs [11]
SEATTLE	Synthetic data based on specs [24]

Table 5: Datasets used

implemented on LEAP and a performance and sensitivity analysis. To quantitatively evaluate LEAP, we compare it to an optimistic model we construct for a fixed-function lookup engine for each lookup. We also compare LEAP to PLUG which is a state-of-art programmable lookup engine.

6.1 Methodology

We have implemented the seven algorithms mentioned in Table 1 using our Python API including the multiple algorithmic stages and the associated processing. For this work, we manually translated the code from the Python API into LEAP configuration bits, since our compiler work is on-going and describing it also is beyond the scope of one paper. For each lookup, we also need additional data like a network traffic trace or lookup trace and other dataset information to populate the lookup data structures. These vary for each lookup and Table 5 describes the data sets we use. In the interest of space, omitted details can be found in [23]. To be consistent with our quantitative comparison to PLUG, we picked similar or equivalent traces and datasets. 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 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 [25] 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 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 [34] overestimate. Our estimate of the FxFu area could be conservative since it does not account for this memory specialization.

6.2 Implementing Lookups

First, we demonstrate that LEAP is able to flexibly support various different lookups. Table 6 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.

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 6: Application Code Statistics

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 7: Latency Estimates (ns)

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

6.3 Performance Analysis

Tables 7-9 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 7 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 latencies 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 8.
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.*

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

Table 8: 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
Small Memory Tile 64 KB			
Total	0.3 mm^2	0.88 mm^2	0.36 mm^2
Memory %	98	35	83
Compute %	2	65	17

Table 9: Area Estimates

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 9. 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.

7. 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 an architecture for efficient soft-synthesiz-able multifunction lookup engines that can be deployed as co-processors or lookup modules complementing the packet processing functions of network processors or switches. 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 implemented 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 typically by $5\times$. 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 will focus on demonstrating 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.

Acknowledgments

We thank the anonymous reviewers for their comments. Support for this research was provided by NSF under the following grants: CNS-0917213. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

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

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

[21] 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*.

[22] 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*.

[23] E. Harris. Leap: Latency- energy- and area-optimized lookup pipeline. Master's thesis, The University of Wisconsin-Madison, 2012.

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

[25] 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*.

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

[27] 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*.

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

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

[30] 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*.

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

[32] 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.

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

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

[35] 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*.

[36] 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*.

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

[38] 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*.

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

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