SWSL: SoftWare Synthesis for Network Lookup

Sung Jin Kim[†] Lorenzo De Carli[†] Karthikeyan Sankaralingam[†] Cristian Estan[‡]
[†]University of Wisconsin-Madison [‡]Broadcom Corporation
{sung,lorenzo,karu}@cs.wisc.edu cestan@broadcom.com

ABSTRACT

Data structure lookups are among the most expensive operations on routers' critical path in terms of latency and power. Therefore, efficient lookup engines are crucial. Several approaches have been proposed, based on either custom ASICs, general-purpose processors, or specialized engines. ASICs enable high performance but have long design cycle and scarce flexibility, while general-purpose processors present the opposite trade-off. Specialized programmable engines achieve some of the benefits of both approaches, but are still hard to program and limited either in terms of flexibility or performance.

In this paper we investigate a different design point. Our solution, SWSL (SoftWare Synthesis for network Lookup) generates hardware logic directly from lookup applications written in C++. Therefore, it retains a simple programming model yet leads to significant performance and power gains. Moreover, compiled application can be deployed on either FPGA or ASIC, enabling a further trade-off between flexibility and performance. While most high-level synthesis compilers focus on loop acceleration, SWSL generates entire lookup chains performing aggressive pipelining to achieve high throughput.

Initial results are promising: compared with a previously proposed solution, SWSL gives $2-4\times$ lower latency and $3-4\times$ reduced chip area with reasonable power consumption.

Categories and Subject Descriptors

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

Keywords

Network processing, Lookups, High-level synthesis, Dynamically-specialized datapath

1. INTRODUCTION

As the demands of Internet services increase, throughput requirements for high performance routers and switches become more and more stringent. At their core, these devices consist of multiple line cards, each including host processors, memory modules and lookup engines. Data structure lookups are are among the most expensive operations on the packets' critical path in terms of latency and power; therefore, developing high performance lookup engines is crucial to enable router performance to scale.

Three main research approaches have been investigated in computer architecture and VLSI design communities. The first consist on SRAM-based algorithmic solution, deployed as software on general purpose processors [12] or even GPUs [20, 31]. Their strength, stemming from the architectural flexibility of general purpose processors, is the ease with which algorithms can be developed, debugged and tuned. However, the lack of specialization of general-purpose CPU negatively impacts performance. Moreover, they exhibit high power consumption, which is the cost of generality [5]. The second approach is hardwired logic design (e.g. [18, 23, 26, 35, 6, 11, 34, 37]). Implementing lookup algorithms with fixed-function hardware maximize performance and minimize power consumption. This however comes at the price of a design cycle that is long and error-prone, and no flexibility. A third option consists of specialized designs implementing programmable hardware accelerators. Such accelerators provide hardware implementation of functionality typically used by lookup algorithms, arranged in a configurable architecture, to achieve the best of both worlds. Examples include [4, 21, 25, 27]. However, as providing both complete programmability and performance is an impractical goal, these specialized approaches still have to prioritize one or the other. Moreover, recent industry trends [3, 33] show that the main cost factors in hardware development are design/verification, followed by developing the companion software. Specialized engines still incur design/verification costs, although amortized. The related software is tied to the architecture; transitioning to a different approach requires rewriting it, wasting a considerable investment.

In this paper, we propose a new approach to design lookup engines called SoftWare Synthesis for network Lookups (SWSL – pronounced Swizzle). SWSL consists in a lookup programming API and a specialized compiler middle layer that generates efficient lookup hardware logic from software. This approach improves the hardware/software development cycle in various ways. First, SWSL is architecture-neutral: lookup implementations are valid C++ that can be compiled for a conventional CPU, or fed to the SWSL compiler and deployed as FPGA or ASIC. The former approach retains flexibility, while the latter prioritizes performance. Thus SWSL eases code reuse. Moreover, software and hardware designs are consolidated: design and verification can be done efficiently in an high-level programming language, reducing the need for a separate hardware verification cycle. Power and area usage are limited, as the specialized logic implements exactly the functionality needed by the application.

The SWSL programming model (§ 3) is dataflow-based and naturally exploits pipelining opportunities present in lookup algorithms [10]. While most high-level synthesis compilers focus on latency improvement through loop acceleration, SWSL leverages the simple, acyclic nature of lookup programs to generate hardware logic for the entire lookup algorithm, performing aggressive pipelining to achieve high throughput. One of the main challenges in synthesizing hardware from software is that the latter is inherently sequential, offering limited opportunities for concurrent ex-

ecution. SWSL employs optimizations to uncover concurrency at basic-block level (§ 4), and increases parallelism through the use of *variable lines*. Each line maintain an independent copy of the execution state of the program, allowing multiple executions to proceed independently in parallel.

We compare SWSL with two previously proposed lookup accelerators, PLUG and LEAP, with promising initial results. SWSL sustains the same throughput as these designs, with reasonable power consumption. Moreover, in comparison with PLUG SWSL gives $2-4\times$ lower latency and $3-4\times$ reduced chip area.

This paper is organized as follows. Section 2 provide background and presents related work both in the fields of lookup accelerators and high-level synthesis. Section 3 presents overview of SWSL, and Section 4 discuss the functioning of SWSL in details. Section 5 presents quantitative measurement and evaluation based on five network lookup algorithms, and Section 6 concludes the paper.

2. BACKGROUND AND MOTIVATION

2.1 Network Lookups

The fundamental task of switches and routers is to determine next-hop information (e.g. an outgoing port number) given some packet data, such as a layer 2 or 3 addresses, or the connection 5-tuple. Making a forwarding decision usually requires searching large data structures of various kinds, depending on the specific algorithm. For example, layer-3 routers use the IP destination address to search IP routing tables; OpenFlow [30] and its predecessor Ethane [9] look up per-flow rules in tables index by layer-3/4 headers. The operation must be performed at line speed and for each incoming packet. As lookups are among the most expensive operations on packets' critical path in terms of latency and power, there is a large body of research, both in academia and industry, on implementing them efficiently.

Software approaches are problematic because network lookups are known to suffer from poor locality: multi-Gigabit routers process packets from tens of thousands of unrelated flows, limiting the effectiveness of caching. Algorithmic lookups approaches rely on specialized data structures, with the aim of minimizing the number of memory references per lookups and the forwarding table size. Examples include [6, 11] for IP lookups and [34, 37] for packet classification. However, because general-purpose CPUs are not optimized for lookup tasks they cannot sustain throughput requirements of large routers.

Ternary content-addressable memory (TCAMs) are a popular hardware-based approach. TCAMs can concurrently compare a search key (including wildcards) with all the entries of a table, implementing hardware bit-level parallelism. Their high throughput comes at a cost: TCAMs are expensive, have low storage density and high power consumption. A different approach consist in implementing algorithmic approaches as fixed-function hardware (e.g. [18, 23, 26, 35]). Such hardware is based on inexpensive RAM and can achieve performance comparable to TCAMs with better power and area usage. However, deploying such designs as ASICs require a long and expensive design and verification cycle, and is hardly flexible.

Significant research effort has focused on flexible lookup engines, with the goal of achieving throughput comparable to ASICs while retaining some programmability. Several proposals use GPUs as packet processing engines [20, 31]. Neither GPUs nor their programming models are optimized for the task; therefore, these approaches have seen little adoption outside academia.

Other lookup engines leverage the observation that – despite their heterogeneity – hardware lookup implementations do have common aspects [10]. Proposals such as [4, 21, 25, 27] provide hardware implementation of functions typically used by lookup algorithms, arranged in a configurable architecture. Similar to GPUs, these engines use specialized programming models; lookup implementations are hardware-specific and cannot be used on different platforms.

Approach	Performance	Efficiency	Design time	Debug/ verif.
Software	Low	Low	Good	Good
Customized Design	High	High	Poor	Poor
HLS	High	High	Good	Good

Table 1: Comparison of design methodologies

We argue that the specialized engine approach does not avoid two significant pitfalls of ASICs: hardware verification costs and excessive software specialization. According to [3, 33], the main cost factors in hardware development are design/verification ($\sim$40%) followed by developing the companion software ($\sim$30%). Designing lookup engines still requires complex and costly hardware/software codesign; lookup algorithms developed for a platform are highly specialized and cannot be ported to different architectures.

In this paper, we propose to tackle these limitations by generating lookup hardware directly from high-level software via *high-level synthesis (HLS)*. No hardware/software codesign is required; verification can be done fully in software using standard debugging tools. Therefore, hardware testing can be minimized to sanity-checking the final design. Moreover the approach is architecture-neutral and reusable, as our API does not assume any special hardware capability. The hardware generated by HLS can be deployed both on FPGAs and ASICs, enabling a trade-off between flexibility and performance. For example an ASIC deployment – sacrificing flexibility – may be acceptable if the target application is well-standardized, e.g. layer-2 or -3 forwarding.

In the next section we discuss the advantage of software- and hardware-based solutions, motivating the need for HLS techniques that can integrate the benefits of both.

2.2 High-Level Synthesis

Implementing functionality in software differs significantly from deploying the same functionality as FPGA or ASIC. In the former approach, algorithms are expressed in a high-level programming language which is compiled for the target platform. In the latter approach, the developer specifies an implementation of the functionality using and hardware description language (HDL) such as Verilog or VHDL. Such description is then synthesized to hardware. The two approaches present different trade-off in several important aspects.

Performance: A HDL implementation can achieve higher parallelism compared to a high-level programming language. Without multi-thread programming, a microprocessor can only execute sequentially. However, the logic in the HDL contains only the data path and simple control logic elements and these elements can be easily arranged to run in parallel for high throughput. Moreover, in hardware much functionality can be assembled to run in a limited number of clock cycles. For these reasons, an HDL design can be more than one hundred times faster than software running on a microprocessor [32].

Efficiency: There are several reasons why the HDL approach can achieve superior efficiency in system design. Because hardware can perform the same computations in fewer cycles, FPGAs and ASICs can often run at a lower frequency than a microprocessor. There-

HLS	Fine Grained	Coarse Grained	Hybrid	SWSL
Config. cell	Logic Gates	ALU, Register, Memory	Reconfigurable Array	Logic Gates
Target Apps	Loops	Loops	Loops	Lookup request
Profiler	Yes	Yes	Yes	No
Output	HDLs	Config. file	Config. file	HDLs
Compilers	ROCCC [17], LegUp [8], NEOSII [28]	PipeRench [14], RaPid [13]	GARP [7]	SWSL

Table 2: High-Level Synthesis Approaches

fore, they tend to consume in tens of watts, while microprocessors consumes more. Moreover, an HDL design provides less architectural overhead, since – contrarily to microprocessors – does not include potentially unnecessary components [2]. According to [32] hardwired logic can be more than one hundred times smaller than an equivalent software implementation.

Design Time: Directly designing using a HDL is notoriously complicated and painful. The designer must take into account both the high-level algorithm and low-level hardware considerations, resulting in a complex, error-prone implementation. According to Kathail, design and verification effort varies depending on the implementation approach [24]. Implementations must go through several steps, with the first being a reference software-only implementation and the final the HDL logic. The HDL implementation alone consumes on average several engineering years. Conversely, software-based approaches allow rapid prototyping and optimization.

Debugging and Verification: High-level programming languages enable rapid prototyping and debugging. Hardwired design with HDL does not provide easy debugging methods, so engineers rely on the output signal from simulators like Modelsim. This is one of the primary reasons for which software is chosen over hardwired logic, even though the latter gives superior performance and efficiency.

High-level Synthesis (HLS) is a set of techniques for generating HDL from algorithms expressed in a high-level programming language (typically C or C++). The goal is to achieve the performance and efficiency of the hardware approach and the ease of design/verification of software. Table 1 compares HLS with both software and hardware approaches under the aspects discussed previously. Table 2 summarizes proposed HLS approaches, including SWSL. Previously proposed approaches focus on loop acceleration to reduce latency. Their design is based on finite-state machines; throughput is limited because a new input cannot be processed until the previous one has finished and the state machine is ready. In general, these approaches are not a good fit for lookup algorithms, which tend to avoid loops by design and focus on throughput more than latency. Besides the approaches in 2, Bluespec and SystemC provide HLS capability via specialized syntax on top of high-level languages [1, 16]. Such specialization limits code reuse.

These considerations motivate the need for a new HLS compiler targeting lookup engine generation. Our proposed approach, SWSL, leverages the simple and loop-free structure of lookup algorithms to derive optimized, throughput-oriented hardware implementations. In addition, SWSL requires no profiler because it generates hardware logic for the whole lookup pipeline. In other words, SWSL can be considered as a SWSL chip generator of lookup engines [33].

3. SWSL

SWSL generates hardware logic descriptions from C++ programs using HLS techniques. It consists of two components: an hardware-agnostic dataflow-based programming model specialized for lookup algorithms, and a compiler middle layer that converts programs to Verilog.

SWSL is based on the observation that lookup algorithms can be decomposed in simple algorithmic steps, each accessing its own private state. The SWSL programming model enables the developer to specify the lookup algorithm as such a pipeline of steps; the SWSL compiler generates Verilog for each step, and connects the steps to implement the full algorithm. The goal of the compiler is to exploit pipelining to handle a request per clock cycle, achieving throughput equal to system clock. At the same time, to reduce latency, SWSL increases parallelism by executing operations that are not data-dependent in parallel.

3.1 Programming Model

Previous work [10] observed that data structure lookups used by network applications tend to consist of simple steps, each accessing some private state. The SWSL programming model reflects this structure, allowing the programmer to express application as a Data Flow Graphs (DFG). This approach is inspired by the PLUG programming model described in [10]. Each node in a DFG represents an algorithmic step, and includes some simple computation and the associated data. This model is a particularly good fit for SWSL as the application workload is well-partitioned in simple steps.

Figure 1 shows an overview of how SWSL generates lookup hardware from applications expressed in the DFG model. Figure 1a depicts the DFG of a simple application. The first stage computes an hash that is then used to perform lookups in two secondary stages. If both stages return a result, one takes priority over the other. This simplified graph can for example model a router/-firewall, where a high-priority table lists specific flows that must be dropped while a low-priority table holds more general forwarding rules. SWSL takes the functions executed at each stage and generate equivalent blocks of lookup hardware logic (Figure 1b). Finally, each generated logic block is associated with a SRAM module to hold the forwarding table, and all blocks are combined in a single lookup chain (Figure 1c). Blocks in the lookup chain are independent and communicate via on-chip messages.

This approach has two significant advantages. In comparison with a software-only approach, SWSL replaces Von-Neumann style cores with specialized logic, increasing efficiency and performance. Moreover, DFG-based applications express algorithms in a form that is convenient for hardware generation, yet architecture-independent. Indeed, we were able to use SWSL on applications written for the PLUG accelerator, which has a similar model, with no modification (Section 5). We also note that application written for SWSL are in standard C++, and they can be compiled and run fully in software for prototyping/debugging purposes.

From the point of view of the programmer, SWSL poses a series of constraints with the goal of keeping the algorithm hardware-friendly. First, to enable pipelining SWSL requires programs to be loop-free (this condition could be easily relaxed to require all loops to be bounded, enabling static unrolling). To minimize the impact of this constraints, the SWSL programming model provides API primitives to perform common loop-based operations such as bit-counting. Then, SWSL requires accesses to the main lookup data structure to be performed with dedicated API calls, thus being clearly separated from accesses to temporary variables. This enables SWSL to discriminate temporary variables and store them in fast registers, avoiding related memory references at runtime. To

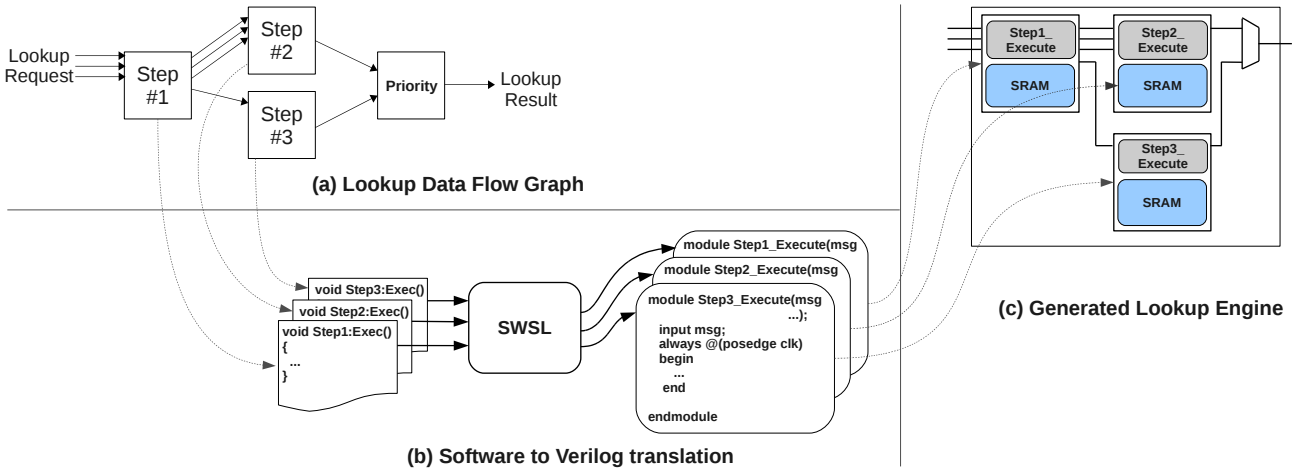

Figure 1: Lookup Engine Generation using SWSL

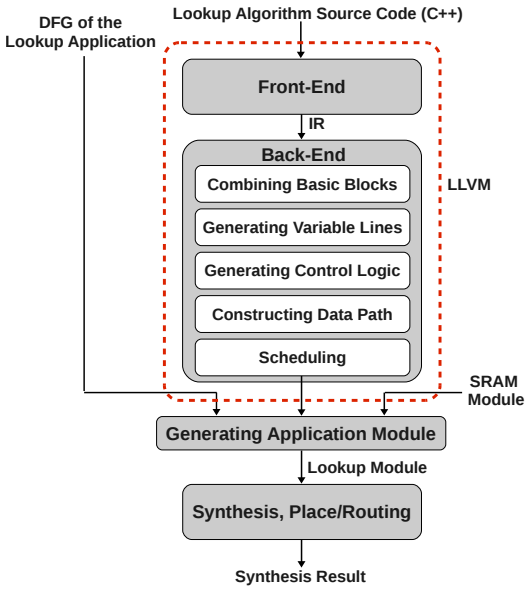

Figure 2: Compilation Process of SWSL

prevent ambiguous memory references to such variables, SWSL does not allow pointer arithmetic. Finally, dynamic memory allocation is not available as this concept cannot be mapped to a static hardware implementation.

In general, we found that such constraints do not limit the expressiveness of the programming model significantly, and lookup algorithms can be naturally implemented with SWSL.

3.2 SWSL Compiler

Algorithmic steps (represented as nodes in the DFG of Figure 1a) are individually converted to hardware logic via the SWSL compiler. This component is implemented as a pass within the LLVM compiler toolkit. LLVM provides the basic compilation infrastructure to parse C++ source code, translate it to intermediate representation and build the program control flow graph (CFG). In the CFG, the program is decomposed in basic blocks - straight lines of code with a single point of entry and one or more exits (e.g. a branch or a switch construct). Arcs represent the control flow, i.e. the possible paths the program can follow when executing. The first transformation performed by SWSL is to restructure the control flow to merge certain basic blocks that can be executed in parallel (specifically, basic block that compute multiple conditions evaluate by a branch instruction).

In general the structure of the dataflow graph, where each algorithmic step is represented as an independent node (Figure 1a), offers some opportunity for pipelining. However, the gain in throughput is limited as the graph coarsely subdivides the algorithm in a limited number of steps, each of which can take tens of cycles to execute. To improve throughput, SWSL further internally pipelines each step. In general, the structure of the computation may not naturally lend itself to pipelining. SWSL circumvents this problem by replicating the temporary state associated with the computation. SWSL instantiates multiple buffers called *variable lines*, each capable of holding inputs, intermediate results, and output used/-generated by each step. By using different variable lines, multiple computations can proceed independently. This enables SWSL to pipeline the logic at the basic block level.

After generating variable lines, SWSL creates the control logic that determines which execution path is followed at runtime (such logic will decide, for example, which step must be activated after a branch condition). Then it generates the actual hardware datapath implementing the functionality described by the software. In general, the resulting hardware will have multiple execution paths; the actual path followed during each executions will depend on the result of branch conditions. As SWSL organizes the logic in a pipeline, it must ensure that the number of steps is constant same regardless of which path is followed (e.g. the number of step must not change depending on which side of a branch is taken). SWSL performs an scheduling step that inserts additional delay buffers to "pad" all paths to the same length.

Figure 2 summarizes the overall SWSL compilation process. After been parsed by LLVM, each step in the original algorithm (node in the DFG) goes through SWSL, which generates hardware logic. At the end, a SRAM block is added to the module to store the subset of the forwarding table associated with the computation. At the synthesis and place/routing step, the generated top level design is synthesized using a general synthesis tool (our implementation uses the Synopsys design compiler).

4. IMPLEMENTATION OF SWSL

In this section we discuss the specifics of the SWSL compiler passes outlined in § 3.2.

4.1 Combining Conditional Basic Blocks

Conceptually a microprocessor executes instructions in sequence; accordingly, the CFG presents a sequential view of the code, specifying the ordering between instructions. However, hardware logic is not limited to sequential execution and can perform operations in parallel as long as they are not data-dependent. Based on a representative set of lookup algorithms, a significant source of potential parallelism lies in branch instructions with multiple conjunctions (e.g. *if (A and B and C) then ...*). In the lookups we considered (§ 5) such conditions are usually very simple, but result in code with multiple nested branches (e.g. *if (A) then if (B) then if (C) then ...*). While evaluating each condition separately is efficient in software, it is unnecessary in hardware. SWSL detects occurrences of nested branches and merges the corresponding basic blocks, computing the branch condition in a single pass. To collect such basic blocks, SWSL uses the concept of Merged Basic Block (MBB). Informally, a MBB is a set of branch conditions with no data dependencies between them and the same successors.

Algorithm 1 summarizes how basic blocks are combined, and Figure 3a-e presents an example taken from an Ethernet forwarding algorithm. The code itself is relatively simple, but its control flow graph has some basic blocks which can be combined as a group (`if (msg_vec_in ...)`). Original source code is in Figure 3a; the initial CFG generated by LLVM for the branch condition is in Figure 3b. Gray basic blocks in Figure 3b-e represent those having conditional expressions (LLVM "branch" instruction).

At the beginning, each basic block in the CFG is encapsulated in its own MBB and the set MBB_SET includes all of them. At each iteration, the algorithm considers all the MBB in the set, looking for merging opportunities. Mergeable MBBs have the following properties:

1. A target MBB must have two successors.
2. A target MBB must have only one predecessor.
3. A set of predecessor's successors of a target MBB must intersect MBB with a set of successors of a target MBB.

Algorithm 1 Combining Conditional Blocks

$MBB\_SET \leftarrow Set\ of\ All\ MBBs\ in\ a\ Target\ Code\ Block$
$OLD\_MBB\_SET \leftarrow NULL$
while $MBB\_SET \neq OLD\_MBB\_SET$ **do**
  $OLD\_MBB\_SET \leftarrow MBB\_SET$
  while $MBB\_SET \neq \emptyset$ **do**
    $T \leftarrow Pick\ a\ MBB\ from\ MBB\_SET$
    $Eliminate\ T\ from\ MBB\_SET$
    $numOfSucc \leftarrow Get\ \#\ of\ T's\ Successor$
    $numOfPred \leftarrow Get\ \#\ of\ T's\ Predecessor$
    if $numOfPred == 1\ \&\&\ numOfSucc == 2$ **then**
      $PRED \leftarrow T's\ Predecessor$
      $A\_SET \leftarrow All\ Successors\ of\ PRED$
      $B\_SET \leftarrow All\ Successors\ of\ T$
      if $(A\_SET \cap B\_SET) \neq \emptyset$ **then**
        $Combine\ T\ with\ PRED$
        $Update\ successors\ of\ PRED\ with\ T's\ successor$
  $MBB\_SET \leftarrow All\ Combined\ MBBs$

Conditions 1-2 guarantee that the target is a proper branch node. Condition 3 verifies that both branches have a common successor. For example, in Figure 3b, MBB0 and MBB1 share BB3 as a common successor, and can therefore be merged (Figure 3c). The algorithm cycles through the set of MBB (MBB_SET) until the set is final (i.e. no changes are made during an iteration), as in Figure 3d. Figure 3e equalizes execution paths to ensure correct pipelining, as explained in § 4.2.

We note that programs may in general presents other opportunities for parallel execution besides nested branches; however we find that our simple approach works well in practice and significantly reduces the number of steps in the generated hardware logic.

4.2 Generating Variable Lines for Pipelining

Each step in a lookup algorithm uses variables to represent inputs, outputs and temporary intermediate results. If the algorithm is executed as software on a conventional Von-Neumann architecture, these variables are located in memory and/or registers, so they can be retrieved during execution.

A straightforward hardware implementation of this approach would generate a buffer for each variable in the code. However, this approach would prevent pipelining, for the following reason. A lookup triggered by an input message could access a variable while a previous, ongoing lookup tries to access the same variable at a different point in the program. As accesses refer to different lookup executions, this situation would bring race conditions and unwanted results. To circumvent this problem, SWSL stores variables in multiple *variable lines*. Each variable line is reserved for a given program execution to avoid race conditions. An index counter is provided to each pipeline stage to enable it to access the correct variable line for the current execution. The index counter is incremented whenever a new lookup starts and its value is cascaded to each pipeline stage. By cascading the counter values, each stage reads/writes from/to independent variable lines. The number of instantiated variable lines is the same as the length (in number of stages) of the critical path of the generated hardware pipeline. In this way the hardware never runs out of variable lines, enabling one new lookup per clock cycle. This approach also makes flow control unnecessary, as variable lines also hold input message(s) for each given execution. Therefore, every time a message arrives there is a variable line available to buffer it.

Figure 3f shows the organization of variable lines from the source code in Figure 3a. Once the code is translated to LLVM IR, variables can be readily identified as they are allocated using the LLVM instruction "alloca". SWSL collects the variables and generates variable lines as a set of buffers, each large enough to store all variables.

As outlined above, an index counter must be propagated from each pipeline stage to the next to ensure that the correct variable line is accessed. Since SWSL pipelines the logic at the basic block level, during execution each basic block has the responsibility to set the variable line index of its successor(s) to the correct value.

In general, it is possible for a basic block to have multiple predecessors executing concurrently and trying to set conflicting counter values. For example, in Figure 3d, MBB0 and BB4 are both predecessors of BB3. Suppose two consecutive lookups are submitted to the hardware. While BB4 is being executed as a result of the first, MBB0 will be already processing the second. As a result, at the end of their execution both will try to set the index counter of BB3 to conflicting values. To avoid this situation, SWSL equalizes the CFG by inserting dummy basic blocks, thus ensuring that all execution paths have the same number of stages. In Figure 3e, MBB0 is prevented from directly setting BB3's counter by the instantiation of a dummy basic block. This guarantees that at each cycle only one predecessor will set BB3's counter.

Algorithm 2 shows how SWSL instantiates counters for each basic block, and links each counter with the successor that must receive its value. The algorithm is simple and works regardless

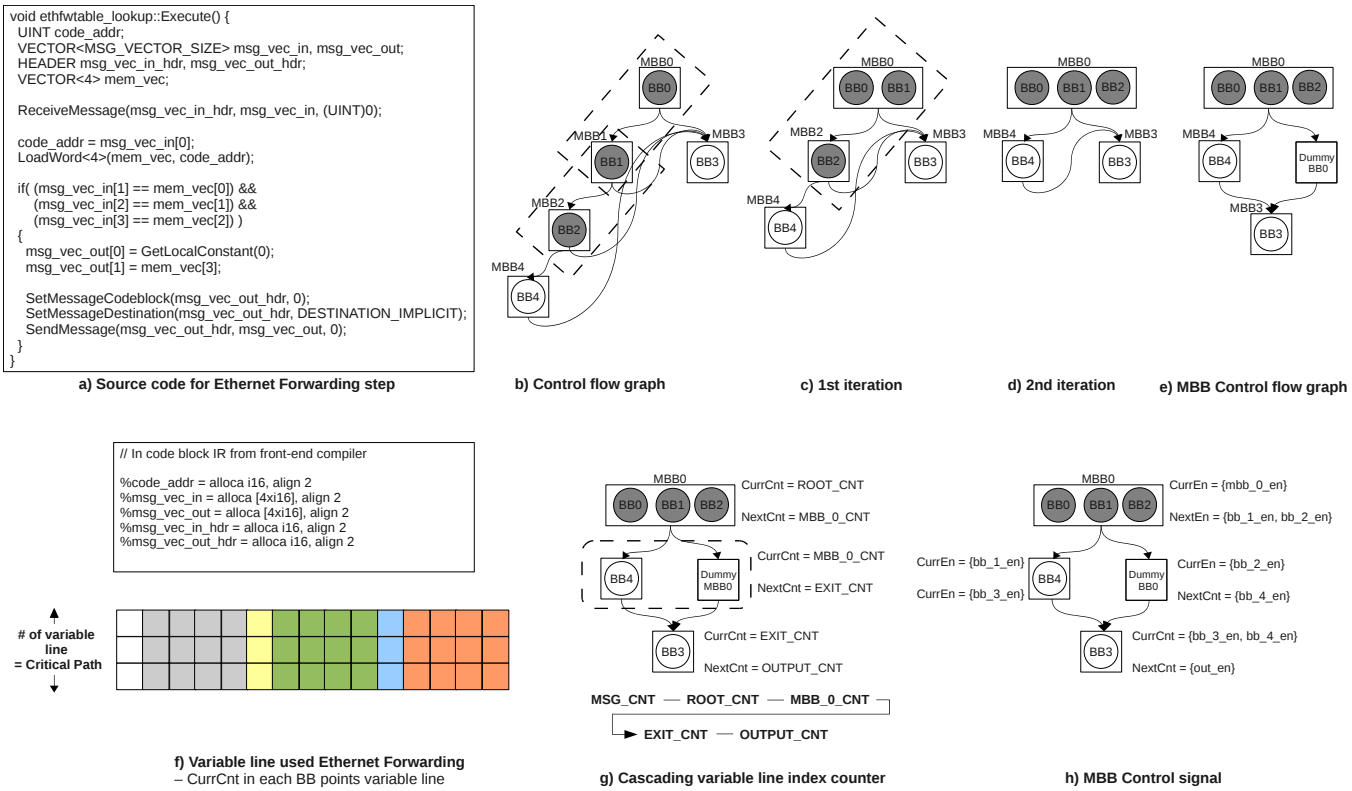

Figure 3: Ethernet Lookup and its Control Flow Graph

Algorithm 2 Cascading Variable Line Index Counter

1. $MBB\_SET \leftarrow Set\ of\ MBBs\ in\ CFG\ except\ Root\ and\ Exit\ MBBs$
2. $MC \leftarrow 'MSG\_CNT'\ for\ Input\ Message\ Data\ Counter$
3. $RC \leftarrow 'ROOT\_CNT'\ for\ Root\ MBB\ Counter$
4. $EC \leftarrow 'EXIT\_CNT'\ for\ Exit\ MBB\ Counter$
5. $OC \leftarrow 'OUTPUT\_CNT'\ for\ Send\ Message\ Counter$
6. $For\ Root\ MBB\ 'R'$
   $R \rightarrow CurrCnt\ = RC$
   $R \rightarrow NextCnt\ = new\ 'BB\_(i)\_CNT'$
   $i++$
7. $For\ Exit\ MBB\ 'E'$
   $E \rightarrow CurrCnt\ = EC$
   $E \rightarrow NextCnt\ = OC$
   $i++$
8. $B \leftarrow\ Pick\ a successor\ of\ Root\ MBB$

$// Pred(B)\ -\ B's\ predecessor\ MBB$
$// Succ(B)\ -\ B's\ successor\ MBB$

repeat
  $Erase\ B\ from\ MBB\_SET$
  if $Pred(B) \rightarrow NextCnt \neq NULL$ **then**
    $B \rightarrow CurrCnt\ =\ Pred(B) \rightarrow NextCnt$
  else
    $B \rightarrow CurrCnt\ =\ new\ 'BB\_(i)\_CNT'$
  $i++$
  if $Succ(B) \rightarrow CurrCnt \neq NULL$ **then**
    $B \rightarrow NextCnt\ =\ Succ(B) \rightarrow CurrCnt$
  else
    $B \rightarrow NextCnt\ =\ new\ 'BB\_(i)\_CNT'$
  $i++$
  $B \leftarrow\ Pick\ a successor\ of\ Root\ MBB$
until $MBB\_SET\ =\ \emptyset$

of the order in which basic blocks are considered. The algorithm initially instantiates input and output counters (i.e. counters received from the previous step and forwarded to the next step). It then define, for each basic block in the CFG, the counter associated with its execution (*CurrCnt*) and the counter that must be set next (*NextCnt*). For each pair of basic blocks $BB1$ and $BB2$, the algorithm ensures that if $BB1$ precedes $BB2$, then $NextCnt(BB1) = CurrCnt(BB2)$. Figure 3g shows an example of correct assignment of counters for the CFG of Figure 3e.

4.3 Generating Control Logic

The main role of control logic in SWSL is to let the generated Verilog code decide which MBBs are executed. Therefore, each basic block must, at the end of its execution, generate an enable signal for the basic block(s) that follow. For each basic block, SWSL defines two signals: current-enable (*CurrEn*), that causes the current basic block to run, and next-enable (*NextEn*), that triggers execution of the basic block's successor(s). Similarly to variable line counters, if $BB1$ precedes $BB2$, then $NextEn(BB1) = CurrEn(BB2)$. The algorithm that instantiates enable signals is conceptually equivalent to the algorithm that instantiate variable line counters (Algorithm 2), with a few caveats. If a basic block has multiple successors as a result of a branch condition, at runtime it will only generate the enable signal for the taken successor. If a basic block has multiple predecessors its $CurrEn$ signal is generated by OR'ing the $NextEn$ signals of all its predecessors. This ensures that the basic block is activated regardless of the path through which is reached.

Figure 3h shows enable signal instantiation for the CFG of Figure 3e. As can be seen, MBB0 can generate two $NextEn$: one activates BB4 and one activates Dummy BB0. During each execu-

tion only one successor is activated, depending on the result of the branch instruction in MBB0. Also note that BB3's can be activated either by BB4 or from Dummy BB0. Its enable signal $CurrEn$ is defined by OR'ing the enable signals of its predecessors.

4.4 Data path in MBB

Once the previous steps have been performed, SWSL generates a Verilog behavioral model implementing the functionality of the software provided as input. To do so, for each basic block SWSL constructs an equivalent dataflow-graph describing how values flow from instruction to instruction. The dataflow-graph is then converted to an hardware datapath.

Care must be taken to handle data dependencies. SWSL must identify all definitions and uses of any given variable, even when the variable is not referenced directly but passed through a memory reference. To do so, LLVM identifies all occurrences of the LLVM "store" instruction, and saves the related memory address references. It then searches for "load" instructions and determines, for each "load", on which "store" it depends. To see why this simple approach works, recall that the SWSL programming model does not allow pointer arithmetic nor dynamic memory allocation. Moreover, the lookup data structure is never accessed directly, but only through dedicated function calls. Therefore, all "load" and "store" in the LLVM IR are guaranteed to univocally refer to stack-allocated temporary variables.

Once all references to a variable have been determined, SWSL can propagate values from producer to consumer instructions. Values referenced across basic blocks (i.e. across pipeline stages) are propagated by storing them in variable lines. This approach however would introduce excessive latency for values produced and consumed within the same basic block. In this case, SWSL defines a "wire" connection in the datapath, so that the value is directly forwarded from one instruction to the other.

Correctness of the overall datapath is enforced by cascading indices and enable signals as described in § 4.2 and § 4.3 respectively.

4.5 Scheduling

The designs generated by SWSL have two kinds of shared resources: memory blocks and on-chip links between algorithmic steps (Figure 1c). These resources are not overprovisioned. Therefore, SWSL must statically ensure that no conflict arises at runtime due to multiple stages in the lookup pipeline trying to access the same resource in the same cycle. Such situation can happen when two distinct executions of the lookup algorithm follow different paths. By default there is no guarantee that all paths will access a shared resource after a constant number of cycles, potentially leading to inconsistent timing between executions.

Even when conflicts do not arise, inconsistency can be caused by the overall shape of the lookup DFG: for example, in figure 1a and 1c there is no guarantee that step 2 and step 3 will present their results to the priority selector in the same cycle, as they may take different times to execute.

Both problems are similar to the static scheduling issues which arise in the PLUG programming model [10], and are tackled by SWSL in the same way. To prevent conflicting accesses to shared resources, SWSL measures the timing of such accesses across all possible execution paths, introducing delay chains when needed to equalize timing. To ensure that parallel algorithmic steps (such as step 2 and 3 in Figure 1c) execute in constant time, SWSL adds dummy delay basic blocks to shorter steps.

5. EVALUATION

In this section, we present a quantitative evaluation of the feasibility and effectiveness of SWSL as a lookup hardware generator.

First, we discuss whether SWSL can support realistic lookup algorithms. We consider 5 representative applications developed for PLUG (a previously proposed lookup accelerator), covering various network protocols and operations. Results are encouraging: SWSL was able to generate functionally complete lookup pipelines for each application in the set.

We then compare the hardware pipelines generated by SWSL with specialized lookup accelerators (PLUG and LEAP) implementing the same functionality. In all cases, SWSL designs achieve the same throughput, comparable latency and power, and smaller area.

5.1 Methodology

To implement SWSL we used the API we previously developed for PLUG [25], a specialized lookup accelerator. Its API exports a DFG-based programming model, and provides infrastructure for simulating and verifying lookup algorithms in software. We then modified the PLUG toolchain to use the SWSL compiler as a backend. The SWSL compiler itself was implemented as a series of passes for the LLVM compiler toolkit.

As target lookup applications, we choose a suite of three standard lookup algorithms (Ethernet forwarding, IPv4, and IPv6), and one research protocol – Ethane, which is an academic precursor of OpenFlow. To evaluate the flexibility of SWSL we also included DFA-based regexp matching (a widespread primitive used in intrusion detection systems). We note here that the DFA application implements only a lookup in a compressed transition table, not the complete DFA. As SWSL shares PLUG programming API, we use version of these applications previously implemented for PLUG. To generate SWSL designs, the applications were fed to the SWSL compiler, implementing the passes listed in Figure 2. For further details of each application the reader is referred to[25, 10]. We reuse these applications from PLUG source code and code blocks with DFG configuration of applications are directly injected in the SWSL compilation process shown in Figure 2. Details of DFGs for each application can be found in [25].

As terms of comparison we use PLUG and another specialized lookup accelerator, LEAP [21]. The choice of comparing with PLUG and LEAP is motivated by the fact that they represent opposite points within the software/hardware design tradeoff. PLUG adopts a software approach, with its computation engines being conventional Von-Neumann cores. LEAP instead uses arrays of fixed-function hardware blocks. PLUG offer software-like programmability; LEAP has a constrained programming model but achieves near-ASIC performance.

To evaluate the performance of SWSL designs, we used the Synopsys design compiler with 55nm design library and 1 GHz clock frequency. To evaluate SRAM memory installed to each logical page, we leveraged the CACTI modeling simulator [36] with SRAM organized by four 64 KB memory banks for proper comparison to PLUG and LEAP (which use the same memory design).

5.2 Analysis

Effectiveness/Ease of programming: To evaluate the SWSL compiler, we selected a set of 5 applications originally written for the PLUG lookup accelerator. We emphasize that basing the SWSL programming model on the PLUG API is purely a matter of convenience; the SWSL compiler could be adapted to a different API with minimal tweaks (as long as the programming model enforces the constraints listed in § 3.1).

Application	Computation Critical Path	Synchronization	Total Latency
Ethernet forwarding	8	3	11
IPv4	93	35	128
IPv6	175	63	238
Ethane	29	2	31
DFA	22	7	29

Table 3: SWSL Application Latency (ns)

SWSL was able to generate functionally correct lookup hardware for all the applications in our set; none of the applications required changes. According to [25], developing/verifying the PLUG hardware took 6 person-months and developing the applications took 18 person-months. Developing the PLUG-specific compiler took another 6 person-months. In this context the SWSL approach would have made the hardware development cycle largely unnecessary, potentially reducing design time by up to 20%.

Throughput Analysis: Both PLUG and LEAP can achieve a throughput of 1 lookup per cycle with a 1 GHz clock frequency. We verified that SWSL is capable to achieve the same throughput.

Latency Analysis: Table 3 presents the latency of each SWSL-generated application (in ns) in the "Total Latency" column. The latency is further decomposed in the component due to the length of the (hardware or software) critical path, and the latency introduced by the scheduler for synchronization purposes (§ 4.5). The synchronization overhead is relatively high, contributing up to 1/3 of the overall latency. However synchronization is crucial, as it guarantees conflict-free execution, enabling pipelining.

Table 4 further compares the latency of SWSL-generated applications with the same applications deployed on PLUG and LEAP. SWSL achieves a significant latency reduction in comparison with PLUG, for three reasons.

First, SWSL exploits more parallelism than PLUG. SWSL generates merged basic block, computing multiple branch conditions in a single 1-cycle pass. As branch conditions tend to be on the computation critical path, parallelizing their computation directly decreases the overall application latency. Conversely PLUG is based on the conventional Von-Neumann architecture, and executes algorithms in software. In this context branch conditions have to be evaluated sequentially.

Second, the PLUG API offers specialized high-level operation, each of which is translated to several atomic instructions. For example, the PLUG *SendMsg* call is used to forward intermediate results between algorithmic steps. At compile-time, a single *SendMsg* is converted to multiple instructions that copy values to special-purpose network registers, construct the message header and send the message on the on-chip network – requiring 5 cycles. Instead, SWSL can implement the operation directly as efficient hardware and execute all data movements in a single cycle.

Finally, there is no communication overhead in SWSL. PLUG cores are organized in a matrix; the on-chip network allows arbitrary communication patterns between cores. This requires all communication to be mediated by on-chip routers. PLUG employs point-point links and XY routing, making communication latency dependent on the distance between cores. As SWSL specializes the hardware for a single algorithm, communication does not need to be flexible. Stages are connected directly via wires (Figure 1c), making communication latency neglectable.

SWSL has still higher latency than LEAP. LEAP can achieve minimal latency because of its optimized programming model. A LEAP computation engine is an array of specialized hardware units connected via a crossbar, resulting in near-ASIC performance. However, such performance comes at a price in terms of ease of development and generality. Programming LEAP involves routing bits and configuration values between units to implement the desired algorithm. LEAP exports a specialized Python API that alleviate the complexity of this programming model, however applications developed for LEAP are highly specific and cannot be easily ported on other architectures. For example, a LEAP programmer must explicitly ensure that the format and bit width of inputs and intermediate results match the specifications of each functional units. Conversely, SWSL uses a conventional programming model which allows developers to leverage their programming expertise and express applications in a more intuitive form. Moreover, SWSL code is generic and platform-independent form, facilitating code reuse.

Application	SWSL	PLUG	LEAP
Ethernet forwarding	11	55	6
IPv4	128	264	24
IPv6	238	524	42
Ethane	31	100	6
DFA	29	59	6

Table 4: Latency Comparison (ns)

Application	SWSL	PLUG	LEAP
Ethernet forwarding	0.582	0.504	0.392
IPv4	1.753	0.504	0.392
IPv6	5.473	2.331	1.813
Ethane	1.200	0.504	0.392
DFA	0.373	0.756	0.588

Table 5: Power Estimation (W)

Power: For PLUG and LEAP, we estimate power by multiplying the power of a single tile with the number of active tiles configured for a target lookup application. While IPv6 is only configured as 8×8, other lookup applications are configured as 4×4 [25]. From tile configuration, Ethernet forwarding, IPv4, IPv6, Ethane and DFA have 8, 8, 37, 8, and 12 active tiles, respectively. The power consumption of a single tile of PLUG and LEAP is 63mW and 49mW [21]. We adopt this methodology to be consistent with results used in the LEAP work, since we compare to both LEAP and PLUG[1]. For SWSL, we collect power number from Synopsys power compiler with RTL code from SWSL with default activity factor. Table 5 shows total power for each lookup engine approach. Results are mixed, with SWSL consuming slightly more than PLUG and LEAP in most cases, with the exception of the DFA application. This is motivated by the different complexities of each application. DFA consists of a simple algorithm that leads SWSL to instantiate a small amount of logic, leading to low power consumption. IPv4, IPv6 and Ethane are more complex and more deeply pipelined. In this case the generality of PLUG and LEAP lead to smaller power consumption, as the same functionality can be re-used multiple times. Instead, SWSL instantiates dedicated logic for each pipeline stage, resulting in greater power consumption for complex applications. However, it is still a surprising and counter-intuitive result that an application-specific implementation consumes more power than a general-purpose engine. The reason is that our Verilog code generator backend is not as mature as the code-generation for engines like PLUG and LEAP which

[1]In the PLUG papers, per-application power was reported, by considering only the tiles that are active based on individual lookup patterns and code-block activated. In contrast, here we are reporting PLUG power as power consumed by a tile multiplied by number of activated tiles.

Application	SWSL	PLUG	LEAP
Ethernet forwarding	16.897	51.2	33.6
Compute	0.937	18.432	1.68
Memory	15.96	32.768	32.92
IPv4	19.419	51.2	33.6
Compute	3.459	18.432	1.68
Memory	15.96	32.768	32.92
IPv6	56.062	204.8	134.4
Compute	10.177	73.728	6.72
Memory	45.885	131.072	127.68
Ethane	18.384	51.2	33.6
Compute	2.424	18.432	1.68
Memory	15.96	32.768	31.92
DFA	16.354	51.2	33.6
Compute	0.394	18.432	1.68
Memory	15.96	32.768	31.92

Table 6: Area Estimation (mm^2)

can re-use decades of research into instruction-level code generation. Also, as outlined in Section 5.3 there is one known source of significant inefficiency in our compiler. We are currently not using known hyperblock and predication [29] technology to handle control-flow. As a result we have excessively long control-flow paths, which introduce unnecessarily large number of variable lines - each of which consumes significant power. We believe the underlying ideas in SWSL will provide power efficiency after these further engineering challenges are solved.

Area: We collect area for each network application from synthesis result. Table 6 presents area of each network application. From table 6, PLUG and LEAP have relatively larger area size than SWSL.

As PLUG and LEAP have a tiled configuration, with tiles arranged in 4×4 or 8×8 squares, they require significant area. However, the area taken by the computational engines is small; most of each tile's area is used by on-chip memory (PLUG - 64%, LEAP -95%). The computation to memory ratio is also small for SWSL; we found that IPv6 computation area for SWSL is approximately the same as 3 PLUG tiles. An important difference is that SWSL constructs the chip design depending on the structure and computation of the input applications. Instead, PLUG and LEAP designs have a fixed structure, as the same design must support multiple applications (the only degree of freedom is the size – in terms of number of tiles – of the design).

SWSL has high power-area ratio, which means that power consumption of a SWSL-generated design can be high even though its area is small. This can be explained as follows. The central idea of SWSL is to translate software functionality to hardware logic. Each block generated by SWSL implements exactly the action performed by the corresponding software; at run-time, all blocks will be busy performing their respective functions. Instead, PLUG and LEAP provide an array of cores (or, in case of LEAP, specialized computational engines) to which functionality is assigned; the number of cores is overprovisioned to support computing-intensive applications. Therefore, in general not all cores will be active at the same time, leading to less power consumption per unit of area.

5.3 Beyond SWSL

In section 5.2 we analyzed SWSL in terms of latency, power, and area. It yields reasonable result, but its efficiency is below what we had expected. In particular, latency presents room for improvement. The primary cause is that conditional expressions such as nested if/else blocks lead to deep CFGs, which in turn cause latency to run high. We expect that such deep CFGs can be simplified – with corresponding latency decrease – by employing predication, i.e. conditional execution of multiple branch sides in parallel. In particular, predication in SWSL will eliminate conditional MBBs in CFG [22]. Figure 4 presents an example CFG and its relaxation via predication. Due to the large number of if/else statement in figure 4 a) the program has a deep CFG, shown in figure 4 b). The use of predication eliminates the control dependencies between "compare" instructions and the basic blocks following the corresponding "branch" instructions, allowing merging of such basic blocks. In the example, a "compare" instruction generates two predicates and the code sequence that follows is executed based on the result of comparison, as shown in figure 4 c). To hint the advantage of this technique we measured the latency of IPv4 after manually introducing predication. The result show a latency of only 107 ns, which is 18% less than the current SWSL design. Thus, it is possible to eliminate MBBs having conditional expressions, leading to a significant decrease in latency. In addition, predication has a positive influence on power and area: in fact, logic components are also eliminated when the critical path of the CFG is shortened by conditional MBB elimination.

Moreover, SWSL can be adapted to design an acceleration engine for general program workload. At present, SWSL is designed for lookup engines and using properties of SWSL makes possible generate acceleration engines of specific code segment in conventional program. Thus, further enhancements could enable SWSL to generate hardware logic replacing other acceleration engines such as DySER and BERET [15, 19].

6. CONCLUSION

In this paper, we propose SoftWare Synthesis for network Lookup (SWSL) to generate hardware logic from high-level program language. Using a DFG-based application programming model, SWSL enables developers to design fully hardware lookup engines in a short time. At the same time, leveraging CFG analysis and compile-time optimization, SWSL improves performance and achieve reasonable hardware efficiency.

Synthesis results show that SWSL achieves high throughput which is an essential property for network lookup. Latency is better than PLUG, which follows Von-Neumann architecture, while it does not win against LEAP due to the specialized computation design of the latter. Moreover, power estimates show that SWSL's power consumption is comparable to, although slightly worse than, PLUG and LEAP. Finally, SWSL significantly reduces the area requirement, since it implements only the functionality needed by the original lookup algorithm (while PLUG and LEAP must overprovision resources to support arbitrary applications).

Further analysis suggests that performance and efficiency can be improved by enhancing the back-end compiler, for example by introducing predication. Deploying these changes is mostly matter of further engineering the SWSL compiler. Overall, SWSL shows the effectiveness of software synthesis in simplifying the design of complex hardware accelerators, while maintaining a short design time.

7. ACKNOWLEDGMENTS

We thank the anonymous reviewers. Support for this research was provided by the National Science Foundation under the following grants: CNS-0917213 and CNS-1228782. 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.

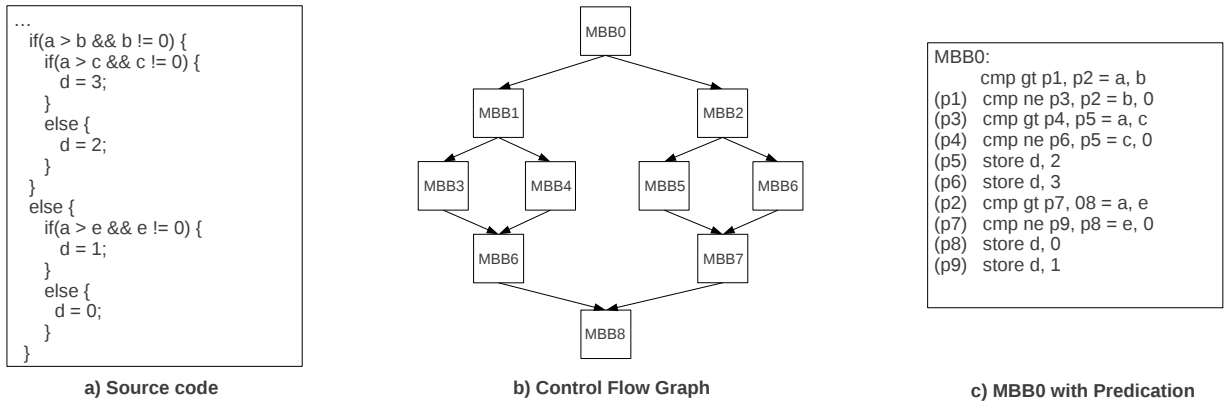

Figure 4: Eliminating MBBs by Predication

8. REFERENCES

[1] Bluespec, http://www.bluespec.com/algorithmic-synthesis.htm.

[2] Comparing and contrasting fpga and microprocessor system design and development https://www.xilinx.com/support/documentation/white_papers/wp213.pdf.

[3] Marc Horowitz. Why design must change (slides), http://www.synopsys.com/Community/UniversityProgram/CapsuleModule/Why-Design-Must-Change.ppt.

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

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

[6] A. Broder and M. Mitzenmacher. Using multiple hash functions to improve ip lookups,. In *INFOCOM '03*.

[7] T. Callahan, J. Hauser, and J. Wawrzynek. The garp architecture and c compiler. *IEEE Trans. Comput.*, 33:62–69, April 2000.

[8] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. D. Brown, and T. Czajkowski. LegUp: An open source high-level synthesis tool for FPGA-based processor/accelerator systems. In *FPGA '11*.

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

[10] 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*.

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

[12] 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*.

[13] C. Ebeling, D. C. Cronquist, and P. Franklin. RaPiD - reconfigurable pipelined datapath. In *FPL '96*.

[14] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer. PipeRench: A coproecessor for streaming multimedia acceleration. In *ISCA '99*.

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

[16] T. GrÃűtker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic, 2002.

[17] Z. Guo, B. Buyukkurt, W. Najjar, , and K. Vissers. Optimized generation of data-path from c codes. In *DATE '05*, 2005.

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

[19] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August. Bundled execution of recurring traces for energy-efficient general purpose processing. In *MICRO 44*, 2011.

[20] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: a GPU-accelerated software router. In *SIGCOMM '10*.

[21] E. N. Harris, S. L. Wasmundt, L. De Carli, K. Sankaralingam, and C. Estan. LEAP: Latency- energy- and area-optimized lookup pipeline. In *ANCS '12*.

[22] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir. Introducing the IA-64 architecture. *IEEE Micro*, 20(5):12–23, 2000.

[23] W. Jiang, Q. Wang, and V. Prasanna. Beyond TCAMs: an SRAM-Based parallel multi-pipeline architecture for terabit IP lookup. In *INFOCOM '08*.

[24] V. Kathail. Creating power-efficient application engines for soc design. *Synfira Inc. Soc Central*, 2005.

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

[28] D. Lau, O. Pritchard, and P. Molson. Automated generation of hardware accelerators with direct memory access from ANSI/ISO standard C functions. In *FCCM '06*.

[29] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *MICRO 25*, 1992.

[30] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.

[31] S. Mu, X. Zhang, N. Zhang, J. Lu, Y. Deng, and S. Zhang. IP routing processing with graphic processors. In *DATE '10*.

[32] C. Rowen and S. Leibson. Flexible architectures for engineering successful SOCs. In *DAC '04*.

[33] O. Shacham, O. Azizi, M. Wachs, W. Qadeer, Z. Asgar, K. Kelley, J. Stevenson, S. Richardson, M. Horowitz, B. Lee, A. Solomatnikov, and A. Firoozshahian. Rethinking digital design: Why design must change. *IEEE Micro*, 30(6):9–24, 2010.

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

[35] 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.

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

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