Designing a Sub-RISC Multi-Gigabit Regular Expression Processor

Andrew Christopher Mihal
Christian Sauer
Kurt Keutzer

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-119
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-119.html

September 26, 2006

Acknowledgement

This work is funded, in part, by the Microelectronics Advanced Research Consortium (MARCO) and Infineon Technologies, and is part of the efforts of the Gigascale Systems Research Center (GSRC).

Designing a Sub-RISC Multi-Gigabit Regular Expression Processor

Andrew Mihal
University of California at Berkeley
mihal@eecs.berkeley.edu

Christian Sauer
Infineon Technologies, Munich, Germany
Christian.Sauer@infineon.com

Kurt Keutzer
University of California at Berkeley
keutzer@eecs.berkeley.edu

ABSTRACT

Increasingly, embedded system designers must exploit application-specific concurrency in order to obtain high performance. Often an application will exhibit several different styles and granularities of concurrency. An average embedded RISC processor is a poor platform when concurrency is a first-class concern. The Sub-RISC paradigm, on the other hand, allows designers to create programmable architectures with application-specific process-, data-, and datatype-level concurrency. This paper describes a Sub-RISC processor that accelerates regular expression matching for network intrusion detection. This processor is lightweight and can be tiled to search multiple packet streams in parallel. Unlike typical application-specific processors, designers are not burdened with assembly language programming. Instead, the language of regular expressions is used as a high-level programming abstraction. Results are shown for ASIC and FPGA implementations using regexp rules from the Snort database.

1. INTRODUCTION

Modern network intrusion detection systems, such as Snort [11], inspect the contents of packets to search for attacks. In order for these systems to work at multi-gigabit line rates, fast pattern matching techniques are needed. General-purpose processors operating at gigahertz frequencies can perform regular expression matching at rates up to a few hundred kilobytes per second [5]. To obtain faster speeds, one must search for ways to exploit the concurrency available in the application.

Fortunately, pattern matching offers many opportunities for parallelism on several levels of granularity:

- *Process-Level Concurrency:* To exploit traditional coarse-grained concurrency, one can process multiple independent text streams at the same time. This can be done against a single set of patterns, or against different sets of patterns.
- *Data-Level Concurrency:* Within a process, multiple steps of a given matching algorithm can be carried out in parallel. One can compare a single text character against multiple pattern characters simultaneously. Conversely, one could search for a given character at several places in the stream. Combinations of these approaches can be used as well.
- *Datatype-Level Concurrency:* At the finest granularity, there is concurrency on the level of individual bits. An implementation can exploit this concurrency by providing hardware support for the mathematical operations and data types used in the application. Pattern matching uses 8-bit comparisons and performs Boolean logic functions on the results.

A great deal of research has explored these areas. Haagdorens et al. [7] focus on process-level concurrency. In this work multiple network flows are processed in parallel against the same pattern database. The target architecture is a dual-processor Xeon with Hyperthreading. Despite hardware support for 4 concurrent threads, only a 16% improvement in throughput is found. Synchronization and communication between processes is a significant bottleneck. Focusing on process-level concurrency alone is not sufficient for a scalable parallel implementation.

To exploit data-level and datatype-level concurrency, FPGAs are an obvious candidate due to their bit-level granularity. Baker and Prasanna build a circuit to accelerate the Knuth-Morris-Pratt algorithm for exact string matching in [1]. This design can process one text character per cycle for a throughput of 2.4 Gbps. The search pattern is programmed into the FPGA's embedded memories. However, network intrusion detection requires hundreds of patterns of the size presented (16 to 32 characters).

The Granidt approach uses a CAM to compare an entire text string against many patterns simultaneously [6]. This scales up to 32 patterns of 20-byte length. After this, the frequency of the design drops and performance suffers.

Several works try to scale further by comparing fewer text and pattern characters at the same time. Baker [2] and Sourdis[12] use partitioning algorithms on the pattern database to avoid instantiating redundant comparators. Cho and Mangione-Smith[3] instantiate comparators for only the prefix of a pattern, and store the suffix in an on-chip ROM. These approaches achieve gigabit throughput rates against hundreds of simultaneous patterns. Bloom filters offer the possibility of scaling to thousands of patterns, if one accepts a small probability of false positives [4].

Unfortunately, these exact string matching approaches are unable to detect all of the attack scenarios found in the Snort database. Full regular expression pattern matching is often necessary. Deterministic finite automata can be implemented in either hardware or software, but require exponential space in the worst case. Nondeterministic finite automata (NFAs) do not have this problem and can be implemented on FPGAs. The original work by Sidhu and Prasanna generates a hard-wired circuit that implements a fixed NFA [10]. These circuits fully exploit data-level concurrency by evaluating all non-deterministic state transitions in parallel in one cycle. The required space is linear in the size of the regular expression and the machine runs in constant time. Franklin et al. [5] achieve between 30 and 120 MHz depending on the size and complexity of the regular expression. This equates to 30-120 MB/sec of throughput.

A common downside to these NFA-based approaches is scalability. Complex regular expressions lead to circuits with long combinational delays which degrade performance. Second, it is not possible to explore different area/performance tradeoffs. There is only one design point which is a complete parallelization of the algorithm. Third, these designs depend on the configurability of the

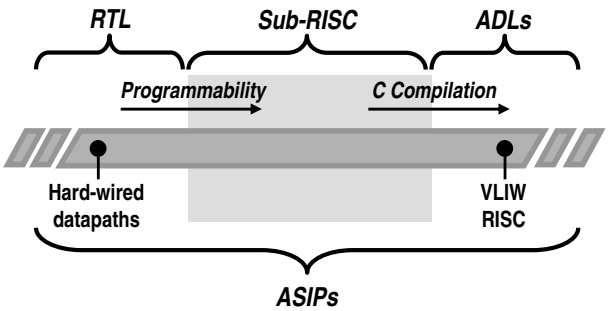

Figure 1: Spectrum of ASIP Architectures

FPGA to be able to change the search patterns. One must synthesize, place and route the design to change the search patterns, and the generality of the FPGA comes at the cost of large silicon area, high power requirements, and low operating frequencies.

In this paper, we propose a new architecture for pattern matching that combines the best features and avoids the shortcomings of these previous works. We apply the Sub-RISC paradigm [9] to create a low-cost, software-programmable processing element (PE) for full regular expression pattern matching. The key to this approach is to support the application's process-level, data-level, and datatype-level concurrency in hardware.

Our NFA PE executes arbitrary non-deterministic finite automata with only a 1-bit wide datapath and 3 function units. With software programmability, we do not depend on FPGA reconfiguration to change the search patterns. Therefore we can realize our architecture as an ASIC for higher throughput and lower cost. Unlike the synthesis approaches, more complex regular expressions correspond to longer programs, not larger circuits. The fixed datapath gives predictable performance in all cases. The PE can also be tiled to make a multiprocessor array for true scalability.

Typical multiprocessors built out of application-specific instruction set processors (ASIPs) are notoriously difficult to program. We show how a NFA PE multiprocessor can be programmed using regular expressions as a high-level programming abstraction to make scalable performance achievable in practice.

In the next section, we describe how the Sub-RISC approach enables matching an application's concurrency requirements at low cost. Section 3 outlines the programming methodology, and the NFA PE architecture is designed in Section 4. Finally, performance results for a Xilinx FPGA-based implementation and an ASIC implementation are given in Section 5.

2. SUB-RISC PROCESSORS

Sub-RISC processors are a class of ASIP. The Sub-RISC design space covers architectures that are programmable, but for which C compilation is not necessary. This space is shown in Figure 1. Software programmability requires control logic that is difficult to specify using conventional RTL hardware description languages. Designers must manually ensure consistency between an instruction set specification and an architectural implementation. Architecture Description Languages (ADLs) provide the right abstractions for solving this problem, but current ADLs focus on RISC-like architectures that run C programs.

The NFA PE does not need to run general C programs, it only needs to execute NFAs. The Sub-RISC approach allows us to design an architecture that breaks the traditional RISC architectural design patterns for the datapath and the control logic, yielding a PE that is a better match for the application's requirements.

It is important to target all three levels of concurrency in the application: process-, data-, and datatype-level. For process-level concurrency, we can create custom control logic that is different from a RISC processor's fetch, decode, and jump logic. The NFA PE evaluates NFA state transitions for streaming character data. It must compute the same set of next state equations for each character. This can be done with a finite-length program that is free of jumps. A Sub-RISC processor can have specialized control logic to match this usage scenario.

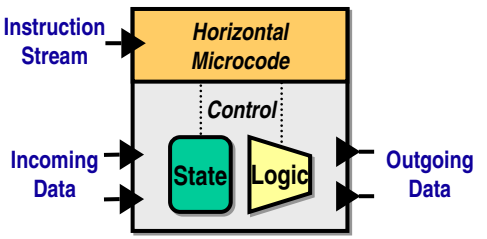

Figure 2: TIPI Control Abstraction

To support data-level concurrency, we can create a tiled multiprocessor architecture that can do several character comparisons and evaluate multiple next state transitions in parallel. To match the application's datatype-level concurrency, we can leave out the traditional RISC 32-bit ALU and register file and instead focus on character-wide comparators and bit-wide Boolean logic functional units. These customizations will provide higher performance at a lower cost than a typical RISC processor.

2.1 Designing Sub-RISC Processors

The Sub-RISC design space is targeted by the TIPI architecture description language and framework [13]. TIPI stands for Tiny Instruction Processors and Interconnect. With this toolset, architects model datapaths structurally. Components such as register files, multiplexors, pipeline registers, arithmetic and logical function units are assembled into novel datapaths that match the application's concurrency requirements.

The atomic state-to-state behaviors the datapath can perform are automatically extracted as a set of *operations* [14]. Operations are a generalization of RISC instructions that describe the programmability of the datapath.

The control abstraction for a TIPI processor is shown in Figure 2. TIPI datapaths are horizontally-microcoded, statically-scheduled machines. A PE idles until it receives a *signal-with-data* message from an external source. The *signal* component of this message is a pointer to a sequence of microcode stored in a program memory. The *data* component contains operands that are made available on the datapath's input ports. A message causes the PE to execute a finite sequence of instructions, after which it returns to the idle state. The program may produce values on output ports which are treated as new signal-with-data messages. These can be sent to another PE in a multiprocessor using an on-chip network, thus continuing the chain of execution.

This control abstraction is a building block for constructing hardware for application-specific process-level concurrency. For network intrusion detection a streaming model is desired. A signal-with-data message will contain a text character and a pointer to a program that computes one state transition of an NFA. These messages will stream into the PE, and the output will be the desired stream of match bits.

TIPI includes code generation algorithms to convert structural datapath models to synthesizable HDL implementations and fast simulators [15]. Architects are not required to write HDL code by hand or specify complex control logic. This enables rapid design space exploration.

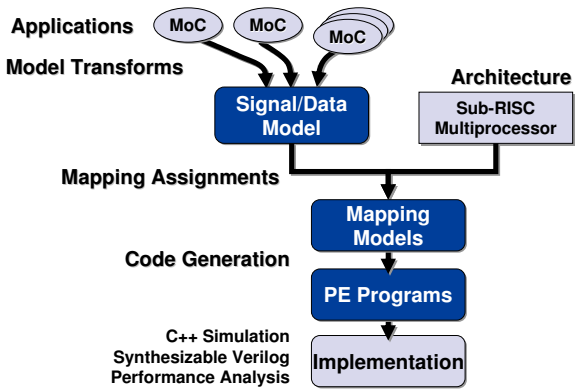

Figure 3: Cairn Y-chart Design Flow

2.2 Deploying Sub-RISC Processors

A common problem with ASIPs is that application experts cannot figure out how to program them. Application domain experts have in mind a concurrent application, but are faced with writing C or assembly language code for individual PEs in a multiprocessor. Here there is a *concurrency implementation gap*. It is difficult and error-prone to translate the application's concurrency into a low-level programming abstraction.

To avoid this problem, the NFA PE is designed to consider deployment as a primary concern. The Cairn methodology provides a discipline for programming Sub-RISC multiprocessors [9]. An outline of this approach is shown in Figure 3.

A core requirement of Cairn is that programmers must use proper abstractions for expressing application concurrency. Traditional languages like C lack the ability to specify process-, data-, and datatype-level concurrency. Inspired by systems such as Ptolemy II, Cairn uses *models of computation* as a basis for application abstractions [8]. Models of computation make it easy for domain experts to make a precise specification of an application's requirements. *Model transforms* are used to replace the abstractions provided by models of computation with concrete implementation details.

For the NFA PE, we will use the language of regular expressions itself as a programming abstraction. A model transform converts a regular expression into a functional model of an NFA that contains explicit concurrency. This NFA model is then assigned to an NFA PE in a multiprocessor in an explicit mapping step. The model is then compiled into an executable program for the PE that computes NFA state transitions in response to signal-with-data messages.

These processes are detailed in the following sections. We start with the transformation of a regular expression into an NFA. Once a formal model of the application is in place, we can then design an architecture that is a good match for the application domain.

3. PROGRAMMING ABSTRACTION

We employ the method of Sidhu and Prasanna as a model transform to convert a regular expression into a NFA [10]. Instead of treating the result as a FPGA circuit, we use it as a bit-level functional model of an NFA.

The first step is to create a parse tree for the regular expression. The leaves in this tree are characters and the nodes are metacharacters. Each leaf and node has a corresponding circuit block. An edge in the tree is a 1-bit bidirectional link between circuit blocks.

Figure 4 shows a simple example for the regular expression a+b$*$. The circuit is akin to a one-hot encoded state machine with one state bit for each leaf in the regular expression. Non-determinism is emulated by allowing multiple bits to be hot at the same time.

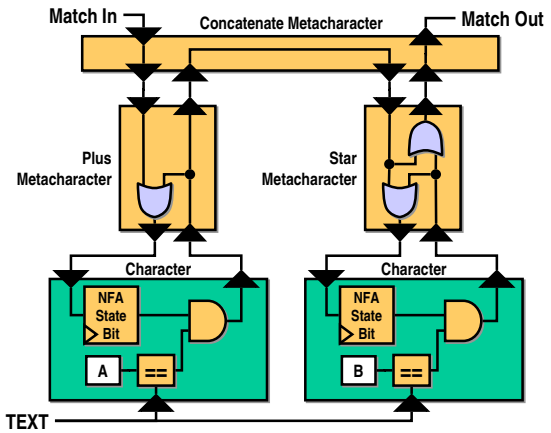

Figure 4: Bit-Level NFA Model for a+b$*$

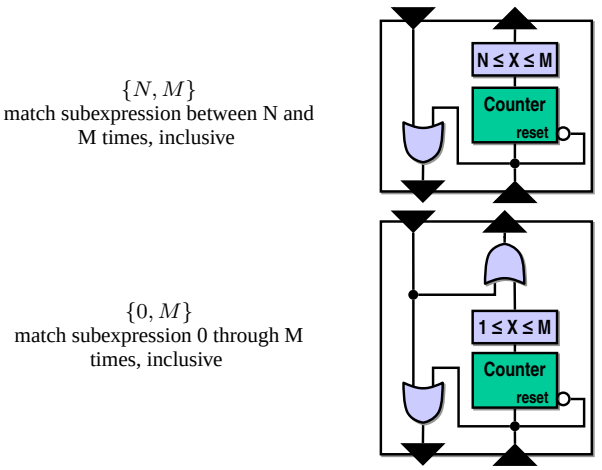

Figure 5: Additional Counting Metacharacters

This bit-level model is an ideal formulation for the Cairn deployment methodology because all of the concurrency in the NFA is explicit. The logic gates and bit-vector wires express datatype-level concurrency. The next state logic can also be interpreted as a dataflow graph that describes data-level concurrency. Process-level concurrency is contained in the sequential behavior of the circuit. In each iteration, the NFA consumes one character from the incoming text stream and computes the circuit's outputs and next state values. The NFA PE will do these same computations with a finite-length program that is free of jumps. One execution of the program corresponds to one iteration of the NFA.

We extend the original work by defining additional metacharacters as shown in Figure 5. Curly braces add additional state to the NFA in the form of counters that record how many times a subexpression matches. If the counter input is true, the subexpression matches in the current cycle and the counter is incremented. The match is propagated up the tree if the count falls within the given range. If the counter input is false, the number of consecutive matches seen is reset to zero.

Additional metacharacters are special cases of the metacharacters in the figure. The $\{N\}$ metacharacter (match a subexpression exactly N times) is $\{N, M\}$ with N equal to M. Likewise, $\{N,\}$ (match N or more times) uses $\{N, M\}$ with M equal to infinity. The ? metacharacter (match 0 or 1 times) uses $\{0, M\}$ with M equal to 1.

If one writes out the next state equations and output equations for a Sidhu and Prasanna NFA by hand, a simple pattern emerges. All

```
match_output  :==  expr
next_state    :==  expr
expr          :==  constant
                 |  state
                 |  compare(state, text, pattern)
                 |  expr or expr
                 |  count_range(expr, counter, N, M)
```

Figure 6: Next State Equation BNF

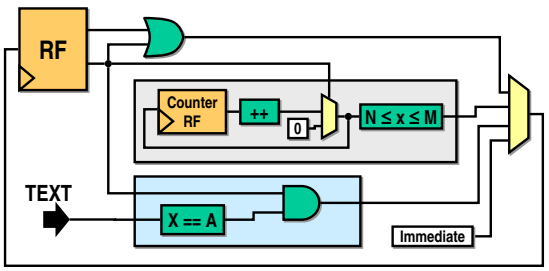

Figure 7: Preliminary NFA PE Architecture

of the signal values in the circuit can be calculated by recursively applying a small set of functions. This can be easily described by the BNF syntax shown in Figure 6. NFA outputs and next state variables are both exprs. An expr can be a constant (0 or 1) or a current state bit. It can also be the result of the conditional comparison function found in the leaves of the Sidhu and Prasanna tree.

There are two ways to build an expr recursively. One is to take the Boolean or of two exprs: an operation found in many metacharacters. The other is to apply the count and range functions found in the curly-brace metacharacters to an expr.

These are the only operations necessary to match a regular expression. We use this knowledge in the next section to guide the design of the NFA PE architecture.

4. NFA PE ARCHITECTURE

A primary goal of the TIPI framework is to enable efficient architectural design space exploration. To demonstrate the utility of this approach we present the evolution of the NFA PE architecture.

The initial version of the NFA PE is shown in Figure 7. This is a simple one-bit wide datapath that contains exactly the function units found in Figure 6. There are two register files. The main one-bit wide RF stores NFA state bits and intermediate computations. The counter RF stores state for curly-brace metacharacters.

It may seem that this figure is a simplification, but it is in fact a complete TIPI architectural specification. Unlike other frameworks, TIPI allows architects to leave many aspects of the control logic unspecified. In this diagram there are no read address, write address, or write enable signals on the register files. The 4-to-1 multiplexor has no select signal. The comparators are missing inputs for the N, M, and A values. In TIPI, these signals are implicitly connected to a horizontal microcode control unit. The operation extraction algorithm discovers the settings that have valid meaning and can be used by statically scheduled software.

There are five valid operations for this datapath. The first is simply the *nop* operation where the machine idles for a cycle. There is an operation that performs a Boolean OR, one that does the count-and-range function, one for the conditional compare function, and one that writes an immediate constant into the register file.

This datapath is also parameterized as follows:

- *Register File Depth*: This controls how many NFA state bits the PE can store. With a deeper register file, the PE can execute more complex regular expressions. In this paper we are using a depth of 256. This is large enough for the most complex regular expression in the Snort database.
- *Counter RF Depth*: Each curly-brace metacharacter in a regular expression requires its own counter. One Snort rule requires 17 counters, the rest use 14 or fewer. PEs with 16 and 32 counters will be mentioned in the results section.
- *Counter RF Width*: This determines the maximum values of N and M in a curly-brace metacharacter. The NFA PE datapath uses type-polymorphic TIPI actors to automatically match this parameter. The largest value found in the Snort rules is 1075, so the counters are 11 bits wide.
- *Character Width*: Normal ASCII characters are 8 bits wide, but here we use 9 bits to allow for ample out-of-band characters (e.g. the $ which matches the end of a stream).
- *Program Memory Depth*: This is the maximum length of the program the PE can run to execute an NFA. We choose a value of 512 for implementation on a Xilinx FPGA, as this is the shallowest BRAM configuration. For ASIC designs, we use a 256-entry program memory.

This datapath architecture is a way to time-multiplex the logic found in an NFA circuit. It can execute any regular expression up to the complexity bounded by the architectural parameters. The more complex the expression, the more datapath clock cycles it takes to evaluate each NFA iteration.

4.1 Design Space Exploration

This first attempt at a PE does a good job at matching the application's datatype-level concurrency. The datapath is only one bit wide and contains exactly the required function units and nothing more. Architects can use the TIPI simulator generator to experiment with this design and measure its performance on various regular expressions. A hardware description can be generated and synthesized to look for hardware-level issues. These experiments reveal some shortcomings that can be easily addressed by making structural changes to the datapath using the TIPI framework.

First, the datapath can benefit from pipelining. TIPI allows users to create irregular pipelines by adding pipeline registers wherever they are necessary. This is an important freedom because Sub-RISC designs often defy the traditional 5-stage RISC architectural design pattern. We added synchronous read ports to the register files, and added a pipeline stage to the inputs of the 4-to-1 mux. The combinational paths through the count-and-range function unit and the text comparator are also broken by pipeline registers.

Second, the immediate field that writes a constant into the register file can be removed entirely by exploiting special cases of the count-and-range function unit. A zero is obtained by executing the count-and-range operation with $N = \infty$ and $M = 0$. These values are reversed to obtain a one. The write enable for the counter RF is disabled in these cases to avoid disrupting counter state.

Third, performance can be improved by adding support for the set comparisons found in regular expressions (the [ ] grouping operator). In the current datapath this must be expanded using the | metacharacter, e.g. [a–c] = (a|b|c). This costs cycles and uses extra entries in the NFA state register file. To solve this we replace the equality text comparator ($t = \mathrm{A}$) with a range comparator ($\mathrm{A} \leq t \leq \mathrm{B}$). To support the [^] syntax, an XOR gate with one input driven from an immediate field in the microcode enables negating the result of a comparison.

For [ ] leaves with multiple items (e.g. [0–9a–f]), we replace the AND gate and the OR gate in the original datapath with logic elements that can perform either Boolean operation. The expression [0–9a–f] becomes $State \wedge ((0 \leq t \leq 9) \vee (\mathrm{a} \leq t \leq \mathrm{f}))$. Similarly, [^0–9a–f] is $State \wedge \neg(0 \leq t \leq 9) \wedge \neg(\mathrm{a} \leq t \leq \mathrm{f})$.

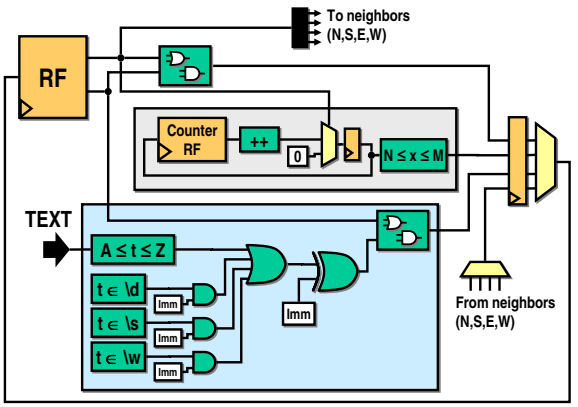

Figure 8: Final NFA PE Architecture

The special characters \d (match any digit), \s (match any white-space character), and \w (match word characters) appear frequently in Snort regular expressions. To speed up these matches, additional hard-wired comparators are added to the text comparison function unit. These are done in parallel with the range comparison and are conditionally included in the result of the compare operation.

Lastly, four pairs of single-bit input and output ports are included for communicating with neighboring NFA PEs in a tiled multiprocessor. These links can be used to communicate match bits off chip. They can also be used to enable mapping a single complex regular expression across multiple NFA PEs to speed up execution.

Figure 8 shows the final architecture. Architects do not have to modify HDL code, update an ISA specification, or verify consistency between an ISA and the datapath implementation to make these changes. Only the structural model needs to be changed. The operation extraction algorithm identifies the changes and propagates this information to the software deployment tool chain automatically. This is an important benefit of the TIPI approach.

5. IMPLEMENTATION RESULTS

To evaluate the performance of our Sub-RISC design, we synthesize TIPI's Verilog output for a Xilinx FPGA and for an Infineon ASIC flow. We program the datapath with regular expressions extracted from the Snort rule database.

5.1 Software Performance

The Snort rules contain 385 unique regular expressions. Cairn provides an automated flow for converting these regular expressions into executable code. First, the model transform described in Section 3 is applied to generate NFA models. A code generation tool takes these models and a netlist of the NFA PE architecture as inputs. For each NFA, a combination of symbolic simulation and Boolean satisfiability is used to find an optimal schedule of TIPI operations that calculate the NFA functions. An architecture-independent code generation approach is used because TIPI datapaths change frequently during design space exploration. Designers cannot afford to rewrite a compiler for each experiment.

The number of processor cycles it takes to perform one iteration of an NFA can be approximated by adding up the cost of each regular expression component, as given in Table 1. Over all 385 regular expressions, there are 7,599 single character, \d, \s, and \w comparisons. The [ ] metacharacters require an additional 2,354 comparisons. The range comparator ($A \leq t \leq B$) proves its worth: if all ranges were expanded to single character comparisons, 14,010 comparisons would be required instead.

The total cost of all 385 regular expressions is 14,058 PE cycles.

Regexp Component	*Cycle Cost*
Single character or \d, \s, \w	1
[ ] with N characters or ranges ($N > 1$)	$N + 1$
[^] with N characters or ranges	N
|, +, $*$ metacharacters	1
{ } metacharacter	2
? metacharacter	2
Concatenate metacharacter	0

Table 1: NFA PE Cycle Counts

The PE spends 70% of its time doing comparisons and 30% of its time evaluating metacharacters. The most complex regular expression found in Snort requires 246 cycles, while the majority take under 40 cycles. Figure 9 shows a histogram of the distribution.

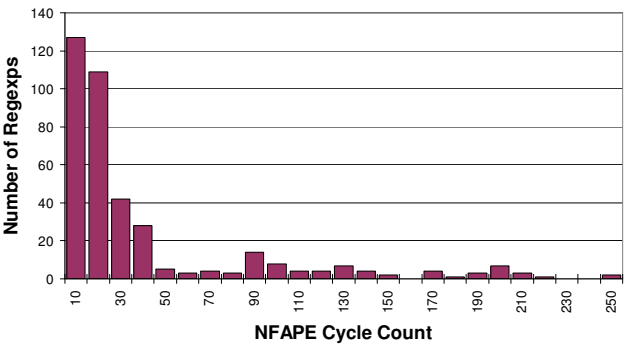

Figure 9: Histogram of Snort Regexp Cycle Counts

In the next sections we construct NFA PE implementations with 256-deep instruction memories. This is large enough for the most complex regular expression. To match all 385 regular expressions in parallel, we build a multiprocessor of 55 PEs and distribute the NFAs across them. Every 256 cycles, the multiprocessor updates all NFAs and consumes one text character.

5.2 FPGA Implementation

Two different Xilinx XC2VP50-5 designs are considered, one with a 16-entry and one with a 32-entry counter register file. Embedded BRAMs are used for the instruction memories. These are configured with the minimum allowed depth of 512 words even though only 256 words are addressed.

Architecture	*Slices*	*BRAMs*	*Clock Rate*
16 Counters	142	3	185 MHz
32 Counters	152	3	185 MHz

Table 2: FPGA Implementation Results

A multiprocessor with 55 PEs consumes 7,810 slices and 165 BRAMs. At 185 MHz, this machine matches one character against 385 regular expressions every 256 cycles for a throughput of 722 KB/sec.

5.3 ASIC Implementation

Both NFA PE variants were synthesized with Synopsys Design Compiler for one of Infineon's 90 nm technologies. For the large instruction memory, a macro was generated. The other memories were synthesized.

We obtain a clock frequency of 980 MHz for typical operating conditions, as shown in Table 3. The critical path depends on external I/O. Since this PE is insensitive to I/O latency and the most critical internal path is 1.042 GHz (Typ/IO) the speed could improve

Architecture	Area (mm^2)	Frequency (MHz)		
		Typ	Typ/IO	Best/IO
16 Counters	0.143	980	1042	1299
32 Counters	0.155	980	1053	1299

Table 3: ASIC Implementation Results

by up to 6% with I/O pipelining. For best operating conditions, the same design can run at 1.3 GHz (Best/IO).

The 55 PE multiprocessor will consume $7.8\mathrm{mm}^2$ and achieves throughputs of 3.9 MB/s at 1 GHz and 5.0 MB/s at 1.3 GHz.

The instruction memory dominates the area of these designs at approximately 75% of the given values. Since these PEs execute branch-free programs, random access to the instruction memory is not required. Instruction compression will be a topic of future work. Another option is to allow multiple copies of the datapath to share the same instruction memory while working on independent packet streams. This is similar to a SIMD architecture. A 16-wide PE can provide 16× more throughput for only 6× more area.

5.4 Performance Comparison

We compare against the work of Franklin, Carver and Hutchings [5] where a complete NFA circuit is implemented on an FPGA. This approach is a completely unrolled design whereas the NFA PE is a completely time-multiplexed design. The authors use 8-bit equality comparisons, so [ ] expressions with ranges are more expensive to implement. For all 385 regular expressions, a total of 20,742 single character comparisons are required. Also, the curly-brace { } metacharacter is not considered.

For an NFA of this size approximately 1.25 slices/character are required for a total of 25,000 slices. This is slightly larger than the XC2VP50 used in this paper. Snort regular expressions do not feature deeply nested metacharacters, so the combinational delay is kept under control. Extrapolating from the published results, a frequency of 50 MHz should be attainable. This machine consumes one character every clock for a throughput of 50 MB/sec.

A multiprocessor ASIC constructed of 16-wide NFA PEs is capable of 80 MB/sec in under 50 mm^2. This savings in area over the FPGA is only one of the benefits of the Sub-RISC approach. First, the speed and area of this design are not dependent on the complexity of the regular expressions. The datapath offers predictable performance in all cases. Second, the fine granularity of the NFA PE makes it possible to target more points in the area/performance design space. The smallest FPGA design requires 25,000 slices and additional throughput comes in increments of this area cost. NFA PEs can be added one at a time for consistent improvements in throughput.

In a network intrusion deployment scenario, the NFA PE multiprocessor will be used alongside a processor that matches packet header fields against Snort rules. In most cases, header processing will determine that a packet stream only needs to be compared against a subset of the 385 regular expressions. To achieve a throughput of 125 MB/sec (for gigabit Ethernet line rate), a single NFA PE at 1.3 GHz can afford to spend 10 cycles per character. If the packet stream can be broken into 16 distinct flows, a single 16-wide SIMD-style PE can spend 160 cycles per character. This is sufficient to process at least four average-size regular expressions (under 40 cycles each as shown in Figure 9). In a multiprocessor, most of the NFA PEs can be configured for the most common regular expressions. With tens of PEs it is therefore possible to achieve multi-gigabit line rates.

6. CONCLUSION

Like most embedded applications, network intrusion detection exhibits multiple flavors of concurrency on several levels of granularity. Designers must exploit this concurrency to obtain high-performance implementations. The Sub-RISC paradigm provides a unique way to accomplish this. Architects can build small, light-weight datapaths that match the application's process-, data-, and datatype-level concurrency. Unlike other application-specific processors, programmers are not forced to write complex assembly code. Strong application abstractions make it easy for domain experts to describe parallel applications. Our results show that by starting with a precise model of a concurrent application, and by providing hardware support for the application's concurrency requirements, one can achieve excellent performance at low cost.

7. REFERENCES

[1] Z. Baker and V. Prasanna. Time and area efficient pattern matching on FPGAs. In *ACM/SIGDA Intl. Symp. on Field Programmable Gate Arrays*, pages 223–232, Feb. 2004.

[2] Z. K. Baker and V. K. Prasanna. A methodology for synthesis of efficient intrusion detection systems on FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 135–144, Apr. 2004.

[3] Y. H. Cho and W. H. Mangione-Smith. Deep packet filter with dedicated logic and read only memories. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 125–134, Apr. 2004.

[4] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep packet inspection using parallel Bloom filters. *IEEE Micro*, 24(1):52–61, 2004.

[5] R. Franklin, D. Carver, and B. L. Hutchings. Assisting network intrusion detection with reconfigurable hardware. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 111–120, Apr. 2002.

[6] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett. Granidt: Towards gigabit rate network intrusion detection technology. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 404–413, London, UK, 2002. Springer-Verlag.

[7] B. Haagdorens, T. Vermeiren, and M. Goossens. Improving the performance of signature-based network intrusion detection sensors by multi-threading. In *Intl. Workshop on Information Security Applications*, page 188, Aug. 2004.

[8] E. Lee. Embedded software. In M. Zelkowitz, editor, *Advances in Computers*, volume 56, pages 56–99. Academic Press, 2002.

[9] A. Mihal, S. Weber, and K. Keutzer. Sub-RISC processors. In P. Ienne and R. Leupers, editors, *Customizable Embedded Processors: Design Technologies and Applications*, chapter 13. Elsevier, 2006.

[10] R. Sidhu and V. K. Prasanna. Fast regular expression matching using FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 227–238, Apr. 2001.

[11] Snort - the de facto standard for intrusion detection/prevention. http://snort.org.

[12] I. Sourdis and D. Pnevmatikatos. Pre-decoded CAMs for efficient and high-speed NIDS pattern matching. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 258–267, Apr. 2004.

[13] S. Weber. TIPI: Tiny instruction processors and interconnect. 2005.

[14] S. Weber and K. Keutzer. Using minimal minterms to represent programmability. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES)*, pages 63–68, Sept. 2005.

[15] S. Weber, M. Moskewicz, M. Gries, C. Sauer, and K. Keutzer. Fast cycle-accurate simulation and instruction set generation for constraint-based descriptions of programmable architectures. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES)*, pages 18–23, Sept. 2004.