Research Statement
Abhishek Mitra (amitra@cs.ucr.edu)

My research focuses on the systematic use of reconfigurable computing devices, primarily FPGAs (Field Programmable Gate Arrays) to accelerate CPU intensive applications. I have accelerated a multitude of applications on FPGA based computing platforms. The applications include Image Processing (DWT), Regular Expression engines, and XML filtering. Since traditional software is written in high-level language, my research also involves coming up with compilation flows that would transform the high level constructs to efficient HDL (Hardware Description Language).

Research Background

Traditional software engineering paradigm involves programming in languages such as C, C++ and Java, which is targeted towards general-purpose processors such as Intel Core 2 Duo. The software is executed in a sequential fashion, the so-called Von Neumann execution model.

Reconfigurable Computing models, utilize large programmable silicon hardware to create datapaths tuned for a particular software loop / algorithm. This datapath does away with load-store and branch instructions and efficiently executes the slow and critical portion of the software. Multiple datapaths and pipelined data paths are generated to maximize the input / output bandwidth available on the reconfigurable computing system. The resulting hardware leads to speedups ranging from 10X to 100X as compared to software executing on general-purpose processors.

My research leverages the Reconfigurable Computing model in the form of FPGA chips which are parallel hardware that can be programmed with specialized HDL. FPGAs are increasingly being made available as co-processors on high-performance computation systems.

The generation of HDL from high-level software language is way too complex for a human developer to handle in a reasonable amount of time due to incompatibilities in the execution paradigm between a traditional CPU and an FPGA. This error prone process manifests itself as the main impediment to a wider use of reconfigurable platforms in high performance computing. Compilation frameworks, thus are a valuable tool for translating traditional high-level description languages to FPGA.

Research Contributions

A. Compiling Regular Expressions to HDL

An example of a widely used inspection and parsing application is regular expression. In regular expression matching process, the input stream is inspected for the existence of one or more member strings of a given regular expression. Regular expressions are usually implemented as one of DFA or NFA in software based systems. DFA implementation is unsuitable for hardware due to the state space explosion of implementing a DFA, and the accompanying memory requirements. NFA based implementations try to mimic parallelism on software based execution by using a stack. Since the software can evaluate only one transition at a given time, a stack-based implementation can store the other transitions for future evaluation at a later time. On the other hand FPGA provides inherent hardware based parallelism, which allows an automata to evaluate more than one state transition at the same time. Therefore NFA based implementations are extremely suitable on FPGAs.

My compilation tool converts a PCRE to HDL via PCRE opcodes. This tool solves a very important limitation towards implementation of PCRE on FPGA. This tools uses the front-end parser of the PCRE compiler, one that produces opcodes based on the regular expression operands, which are in turn obtained from a regular expression. In the original software implementation, a software based PCRE engine running on a processor executes these opcodes. In order to accelerate PCRE, my tool allows one to implement the opcodes on a FPGA based PCRE engine. The implementation of the tool operates within the limitations of FPGA hardware and does not currently support regular expression
operations that could require nesting / recursion. My tool compiles the PCRE opcodes obtained from a regular expression and creates HDL code for each of them. The compilation tool then integrates the hardware opcode blocks with a NFA controller, which is then implemented on a FPGA.

B. Accelerating Regular Expression Matching on SNORT IDS

One area, which has seen a rapid growth in use of regular expressions, is Intrusion Detection System (IDS). Increase in malicious activities using computer networks as a medium, has also resulted in an increased deployment of IDS that scan and intercept network packets containing signatures of such activities. SNORT IDS, one the most popular open source IDS uses PERL compatible regular expressions (PCRE) for its regular expression based rules. Network payload data are streamed to a PCRE engine and is tested with a regular expression deemed suitable by SNORT.

The NFA based regular expression model used by PCRE imposes a high demand on the computation power needed to execute regular expression matches. With current network data links approaching 10 Gbps and higher, software based regular expression engines working on network payloads are unable to cope up with the link throughput while looking for malicious signatures within network packets, especially during an active attack.

Thus my FPGA based implementation of regular expression rules can result in speeding up of an IDS. More so, since a network payload is frequently tested on more than one regular expression rule, I have developed an FPGA based accelerator that can parallelize the regular expression matching, by testing the payload through the required rules simultaneously. Multiple regular expression rules are connected to the same input data stream, allowing a network payload to be tested against multiple rules at the same time.

Multiple input data streams can be matched in parallel banks on a FPGA hardware accelerator. My hardware architecture can work with a load balanced software implementation of SNORT, in which multiple IDS processes generate multiple network payload test requests, which are then processed by multiple instances of hardware based PCRE engines in parallel banks.

With a 128-bit wide input data bus (available on SGI RASC Blade), a single FPGA can cater to 16 8-bit input payload threads. Additional FPGAs can implement many more regular expression banks or can replicate the banks, as deemed necessary.

C. XML Filtering on FPGA

Streaming XML filtering is being used abundantly for publish/subscribe applications (or simply pub-sub). In pub-sub, the message transmission on the internet is guided by the message content, rather than its destination IP address. Selective deliverance of parts of XML documents is obtained by filtering the document through multiple filters described in high-level language such as XPath. In pub-sub systems, each individual subscriber interest is described by an XPath expression. XPath expressions consist of a sequence of XML tags and the relationship between the tags is expressed as axes. XPath infers a tree based navigation over an XML document, and involves a parent-child axis and ancestor-descendant axis.

XPath profiles can be easily converted to PCRE. The XML tags are converted to character match blocks, while the axes define the regular expression sequence. I have used a conversion process to convert XPaths to PCRE. Thereafter I have employed our PCRE to HDL compiler to implement XPath profiles on hardware. The only other addition required would be a XML tag STACK to verify a parent-child axis. Pub-sub involves streaming the same document across multiple XPath profiles, and thus all the subscriber profiles can execute in parallel on a FPGA. Moreover XPath profiles share commonality in their prefix, and hence are optimized to share the common prefix and reduce the area occupied on FPGAs.
Implementing XPath profiles on FPGAs mainly involves implementing character matching blocks to identify XML tags in the input document stream. The character matching hardware block compares sequences of characters from the input stream to a given sequence that define an XML tag. The implemented character matching blocks for the XML tags consist of many redundant blocks, the prime examples being the open tag ‘<’, close tag ‘>’, and end tag ‘/’ characters.

It is possible to simplify the design with a 8-bit stream ASCII decoder. This decoder can be used to decode the 8-bit XML data input into one of each 256 1-bit output per clock cycle. Identifying an XML tag in the input stream would thus involve a simple controller, one that checks for a given sequence of 1-bit decoder symbols. Due to this simplification, the XML filter design using a ASCII decoder is area efficient and it runs at a higher clock rate when compared to the design using character match blocks.

D. FPGA Reprogrammibility and Dynamic Co-Processing

Unlike ASICs and hard silicon devices, FPGAs allow reprogrammability of the hardware. This powerful feature increases the versatility of FPGAs, and increases the number of applications that can be accelerated on the given accelerator. FPGA based accelerators typically utilize a fixed width input / output data interface. The accelerator hardware block connects to this interface and thus is able to send / receive data from the host processor. Interfacing the available library of accelerator blocks to the FPGA interface is a time-consuming and tedious task, which almost always, needs to be taken care of manually. The system designer is left with the task of interfacing each and every accelerator blocks, (usually available as IP cores) to the data interface.

In order to solve this problem, I have developed a tool that automatically generates the communication interface between the data interface and a tightly coupled IP core based accelerator (co-processor) system on a CSoC (Configurable Systems on a Chip) i.e. Virtex-4 FX FPGA. It generates hardware wrappers for the IP core that makes the hardware look like a C function invocation in the host processor source code. Thus a ‘C’ function call in the host processor can change the functionality of the FPGA base co-processor by reprogramming the required bitstream. The SGI RASC library uses a similar mechanism by which an API call can reprogram the Virtex-4 LX 200 FPGA on the blade with a new bitstream.

Partial reconfiguration on the FPGA makes it possible to create a system that allows reconfiguration of pre-assigned parts of the FPGA without affecting the static parts, or inducing a system-wide reset. It is a very powerful tool to overcome the area limitation of a single FPGA platform across multiple applications. The system designer is usually left with the task of generating the interface between static and dynamic regions of the FPGA as required for partial reconfiguration. I have extended the aforementioned interface generation tool to support partial reconfiguration, by generating an interface wrapper that delineates the static and dynamic regions on the FPGA. This feature is useful on CSoCs that implement basic hardware peripherals along with a co-processor on the same reconfigurable fabric.

Another application for utilizing partial reconfiguration on an FPGA are situations that demand quick adaptability. Software defined cognitive radios have used this functionality for a while to adapt the hardware towards external changes in the air interface. There is a limitation to the number of regular expressions that can be implemented on an FPGA. Fast changing network activity scenario, can lead to the IDS selecting among different rulesets in a short period of time. Reconfiguration of the FPGA with the type of intrusion detection engines required at the moment enables adaptability to change with network conditions. I have exploited Partial reprogrammability of FPGA to cater to such dynamic situations, when only few rulesets of regular expression rules changes over short intervals of time.

I have developed an IDS infrastructure on an FPGA accelerated computation system, that can maintain execution through software threads during the brief moment when the FPGA is reconfigured.
With my proof of concept hardware system, the novel design allows partial reprogramming across 16 banks of regular expression rule-sets on an FPGA. This architecture can successfully maintain throughput at 10 Gbps scale even under a range of partial and full reconfiguration scenarios running on a proof-of-concept platform. I have used my PCRE to HDL compiler to compile regular expression based rules to VHDL. Similar rules are grouped together in banks of rule-sets.

I have benchmarked my proof of concept FPGA based regular expression IDS testbed using a thirty-two core SGI Altix 4700 supercomputer with a RASC Blade consisting of two FPGAs. I have implemented 448 different regular expressions in 32 modular rule-sets, on the two FPGAs. Such an architecture is a first of a kind demonstration of an adaptable hardware/software regular expression based IDS. The maximum sustainable throughput of this design is 19.84 Gbps per FPGA. In adaptive mode, the IDS can provide better than 10 Gbps throughput even with 32 partial reconfigurations per minute. This system can also sustain 10 Gbps throughput with four full-reconfigurations per minute per FPGA.

**Future Research Plans**

In the future I plan to do research on investigating the vast pool of compute intensive software applications that can be successfully accelerated on reconfigurable computing platforms. I plan to develop advanced compilation frameworks that would allow software developers to easily and efficiently program powerful reconfigurable computing platforms without delving into the details of the hardware.

Examples of potential applications that I am interested in and could make good use of reconfigurable computing paradigm would be: a) acceleration of analytical queries in databases, b) real-time video transcoding c) genome sequence matching using regular expressions, d) implementing nested regular expressions.

Another reconfigurable computing flavor that has witnessed growth is the use of GP-GPU (General Purpose Graphics Processing Units). These hardware devices provide coarse grained reconfigurability by means of providing multiple programmable fixed data width ALUs. In the future I would want to develop compilation flows that would make good use of these hardware for accelerating regular expression.