A Methodology for Leveraging Reconfigurability in Domain Specific Languages

Maysam Lavasani†, Larry Dennison‡, and Derek Chiou†

†Electrical and Computer Engineering
University of Texas at Austin
{maysam,derek}@mail.utexas.edu

‡Lightwolf Technologies
Walpole, MA
larry@lightwolftech.com

Abstract—Special-purpose hardware can dramatically accelerate an application. However, designing special-purpose hardware is often prohibitively expensive in terms of manpower and time. This paper describes a methodology that uses reconfigurability to enable the efficient compilation of a class of domain specific languages. We present the methodology, a prototype compiler, and a 40Gb/sec network processor designed to be implemented on an FPGA using that compiler.

1 Introduction

All other things being equal, hardware specialized for a specific application will outperform hardware that has not been specialized for that specific application. The higher the level of specialization, the bigger the benefits. However, the cost of designing specialized hardware is often prohibitively expensive in terms of both manpower and time, making it practical only for components with sufficiently high volume to amortize those costs. In addition, the inflexibility of hardware means any changes to the application could make the specialized hardware obsolete.

In this paper, we describe a methodology that generates highly efficient hardware from applications written in a slightly restricted form of C calling a fixed set of accelerators (functional units). The code does not require any hardware-specific tuning or annotations making it essentially identical to writing software. The hardware is generated specifically for the application, making it unlikely that another application could run on it. However the hardware is implemented in reconfigurable logic, specifically FPGAs, to enable application flexibility. The methodology is applicable to a wide range of applications but is especially good at addressing problems with significant data parallelism where the processing of each element of data can vary in execution path, computation time, and resources.

Our prototype application, a network processor, has a throughput of 40Gbps (100M packets per second at 40B packets) implemented in a single Xilinx Virtex 5 TX240T FPGA, which is a medium sized previous generation FPGA. Such performance is slower than the fastest ASIC-based network processors announced by EZChip [1] and Xelerated [15] this year which run at 100Gbps, but is comparable to last years state-of-the-art ASIC-based network processors [13]. Using a large, modern FPGA, our network processor should be at least as fast as the fastest ASIC network processors. The fastest parallelized software router that we are aware of runs at 12Gbps (23.4M packets per second) on 32 Nehalem cores consisting of four systems, each containing eight cores [6]. Our solution supports a single 40Gbps flow with 40B packets, while the software version supports only four flows of 3Gbps, a significantly easier problem. Our system's programmability makes accommodating new protocols/algorithms reasonably easy. Our network processor has other advantages including deterministic performance and resiliency against adversarial traffic.

There are three contributions in this paper:

- A novel methodology that enables domain-specific languages (slightly restricted C plus accelerator functions) to generate efficient hardware.
- A compiler that transforms applications written in such a language to hardware.
- A case study of a 40Gbps network processor that handles IPv4, IPv6, and MPLS packets implemented using this methodology.

2 The Methodology

Our methodology is based on applications written using standard C with calls to a fixed set of functional units defined by the user. This methodology is similar to that of domain-specific languages which have specialized libraries for a specific domain. The programmer views the machine that executes her code as a sequential machine with parallel functional units that are called explicitly and codes accordingly. The code is automatically compiled into synthesizable Verilog that efficiently implements that code. In turn, the Verilog can be compiled for an FPGA or ASIC using standard commercial tools.

Functional units encapsulate common and expensive tasks frequently performed by the application. For example, a content-addressable-memory (CAM), a Trie lookup

(a specialized pointer chaser for networking applications), a large set of shared counters, or a vector multiply engine could be encapsulated in a functional unit. Hardware-implemented functional units are expected to be provided by the user. They can be optimized independently of the application code to improve performance. Any state sharing between data elements is done through the functional units. Our methodology tolerates wildly varying functional unit latency without programmer effort.

2.1 Code

Code is structured as a sequence of *instructions*, where each sequence, including branches, is written to process a single data element. Multiple data elements are processed simultaneously. Support is provided to ensure processing of data elements completes in arrival order when necessary. Each instruction contains up to one call to each of the functional units, random combinational processing, and a computation to determine the next instruction. An instruction cannot be dependent on any computation or functional unit calls performed in the same instruction, but has access to all of the data generated in all previous instructions, as well as the data being processed, through variables that persist between instructions. The next instruction is enabled only after the previous instruction completes, including the return of all responses from all functional units accessed in the previous instruction. Thus, from a software perspective, this system is similar to a lock-step VLIW architecture. However, from a hardware perspective, different functional unit calls from a single instruction complete at different times. Hardware is provided to ensure that all of the functional unit calls from a particular instruction have completed before continuing to the next instruction.

The code is automatically compiled by our compiler into Verilog that specifies specialized hardware "engines" that implement the application specified by the code (Figure 1.) The compiler was written using the ANTLR [2] tool. Each engine executes an instruction, then waits in the functional unit state until all replies from all functional units have returned. The engine then decides the next instruction based on the explicit jump statement in each instruction.

The engines contain specialized hardware that constructs requests to functional units, processes functional unit replies, and computes the next instruction. Engines are tied together though pre-written, configurable infrastructure that distributes requests to functional units, and collects results from functional units that are written back to state accessible by the engines. The infrastructure is not specific to an application domain. The functional units are assumed to be available. A particular application domain will have its own set of functional units.

Our approach is similar to C-to-gates work, such as CatapultC [5] and AutoESL [3] but differs in a few ways. C-to-gates compilers are generally designed to support

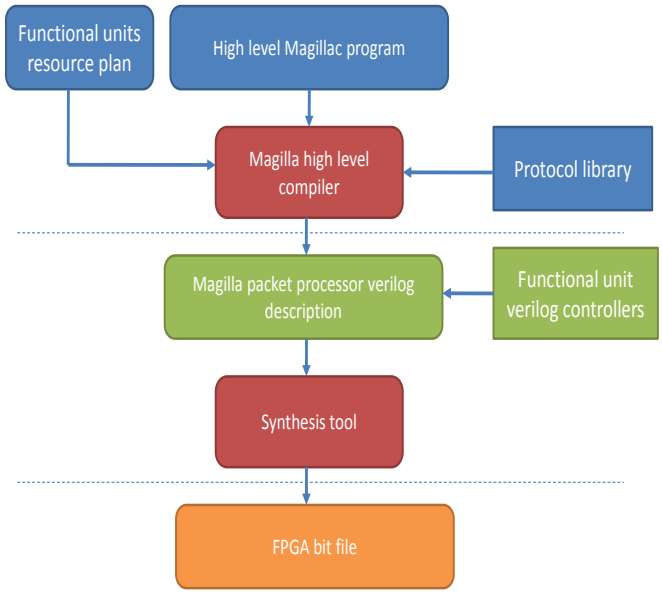

Fig. 1. Magilla translation process

arbitrary C code. However, high quality results generally require stylized code that sometimes includes tool-specific and hardware-specific annotations. Our code, though not full C code, follows the natural flow of the application and requires no hardware specific annotations.

2.2 Compiler Output

The compiler generates the following code for each instruction:

1) Request Builder logic
2) Context Edit logic
3) Jump logic

Request Builder logic computes functional unit request arguments. Context Edit logic updates global variables, using the functional unit replies and state updated by previous instructions. Jump logic determines the next instruction based on the results of the computations in the current instruction. The logic is activated in the appropriate states in the engine state machine.

2.3 Scaling Throughput

Multiple engines allow us to increase the packet processing throughput. Each engine has access to arbitration logic which dispatches the requests to functional units and also returns back the replies to engines.

Although using multiple engines can increase parallelism and help hide the delay of long latency functional units, there are scaling issues. Multiple engines consume multiple sets of hardware resources. In addition, arbitrating between high number of engines is non-trivial.

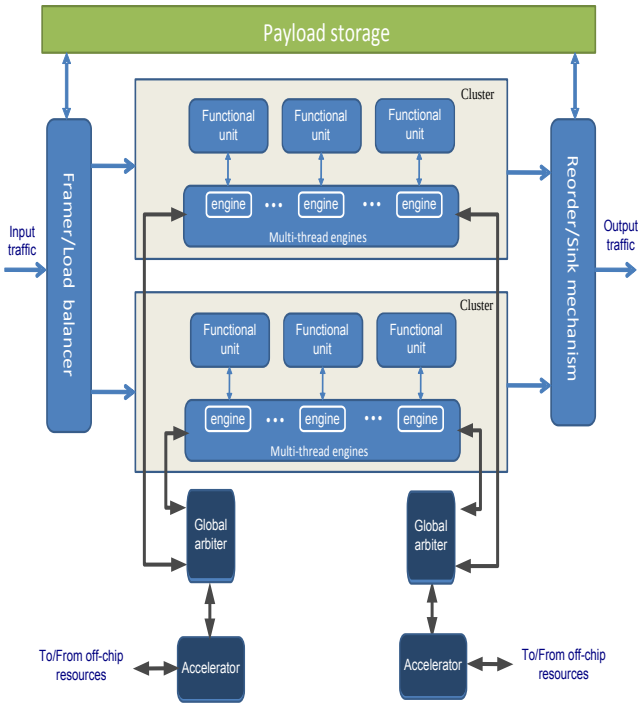

Fig. 2. Multi-cluster Magilla

To make better use of the hardware resources, we multi-thread the engines. Each engine supports multiple hardware contexts and switches threads to tolerate functional unit latency. All threads in the same engine use the same request/reply interface to functional units.

In addition to multiple multi-threaded engines, we also group a number of engines into a cluster and provide multiple clusters. A single cluster shares packet scratch memories to improve utilization of those valuable resources. Figure 2 shows a Magilla system with two clusters.

2.4 Performance Prediction

The performance of such a system is easy to predict, making performance tuning easy as well. Performance is defined by both the functional unit throughput and the demands on the functional units from the application. For a particular application, if the dynamic throughput requirement of each functional unit is less than or equal to the functional unit throughput, full performance will be achieved as long as there is a sufficient number of data elements that can be processed in parallel and sufficient internal state to hold those data elements and the intermediate computations. If the functional unit throughput is not sufficient, performance will be limited by the bottleneck functional unit. Functional units will tend to be implemented with throughput of one to eliminate confusion. Thus, a programmer simply needs to ensure that the available functional units are sufficient for the demands of the application and the desired performance.

2.5 High throughput functional units

FPGA-implemented functional units can be easily added, removed, or modified. It is likely that functional units throughput will be the limiting factor for the whole system. This is due to the fact that the functional units are shared by multiple processing engines. There are several common techniques that can help improve functional unit throughput.

- Higher memory frequencies: In many cases functional units do not do much computation but do access on-chip or off-chip memory. Since it is often possible to operate memory (both on-chip and off-chip) at a higher frequency than the FPGA logic, increasing memory speeds increases memory bandwidth, often resulting in faster functional units.
- Banking: In some cases, banking is possible. For example, using multiple banks for scratch memory can increase overall throughput. This technique was used in the sample IPv4 code described below to increase packet storage throughput.
- Replication: It is possible to replicate the whole functional unit when there are available resources. For example we might use two identical but separate lookup units to double the overall lookup throughput. Of course we need to have enough I/O pins and functional unit bandwidth for such replications if the lookup unit is using off the chip memory.
- Deep pipelining: Pipelining is widely used in lookup units [9] which are based on Trie based data structures. Since most of the off-chip memory technologies are pipelined, this technique allows us to use high capacity memory systems while delivering a high throughput. Deep pipelining increases functional unit delays, but also enables functional units to run at higher frequencies.

3 Example: Routing

Routing is the process of accepting packets, determining what sort of packet they are, determining where the packet should go, and forwarding the packet to that destination. Router programmability has been always considered an important goal since it enables developers to implement their new applications much more easily. New applications include new protocols, deep packet inspection, encryption, measurement and statistics collection, and application acceleration. Network processors are an important part of that programmability.

Routing generally consumes several hundreds of standard RISC instructions per packet. For example, the Intel IXP2800 [7] that is intended to support 10Gb/sec, has 16 cores running at 1.4GHz, for a total instruction budget of 22.4 BIPS or roughly 900 instructions per 40B

packet. The Cisco 40Gbps network processor [13] contains 192 network-customized 500MHz Tensilica cores for a total instruction budget of 96BIPS or roughly 960 instructions per 40B packet. Such high instruction counts result from the many bit manipulation operations, packet data movements, and off-chip accesses data required for every packet. However, even with such high instruction budgets, it is often difficult to achieve the rated performance. Current high-end routers process packets at 40Gbps per interface which translates into more than 100M packets per second assuming 40B packets.

There are several recent projects to add flexibility to packet processing systems. One such project is Openflow [11] that decouples the forwarding decisions (control plane) from the forwarding itself (data plane). This split gives developers some flexibility but provides little support for full programmability of the data plane of a router.

Routebricks [6] uses commodity servers and software-based routers, achieving routing throughput of 12Gbps using 4 Nehalem servers, each with 8 cores. They use several techniques to improve the forwarding capacity of single server including use of the multiple-queue features of NIC cards and batch processing of packets. Both these techniques reduce the per packet processing overhead associated to operating system and general purpose hardware/software package in a commodity server.

PacketShader [12] achieves 40Gbps on a system with two quad-core Nehalem processors, 12GB of memory, two I/O hubs and two NVIDIA GTX480 cards. Packetshader performance is dependent on intelligent NICs that balance load between cores and is currently limited by PCIe performance. Such a system consumes a considerable amount of power where routers are very power sensitive.

Using FPGAs in routers and other communication platforms is widespread because of the short development time compared to ASICs and the available resources in modern FPGAs including logic, memory, and high performance I/O resources [14]. The NetFPGA [10] project provides the FPGA hardware and software infrastructure for packet processing systems, mostly for educational reasons. The fact that NetFPGA has been successfully used as a infrastructure for various networking applications demonstrates the merit of using FPGA as programmable substrate for packet processing systems.

4 Magilla: A 40Gbps Network Processor

We applied our methodology to generate a network processor called Magilla. Our methodology's ability to generate specialized data paths for the particular application, coupled with the ability to support high performance functional units with highly variable latencies make it ideal for network processors.

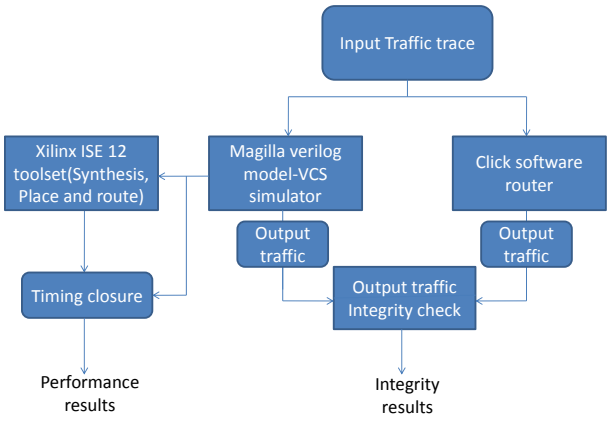

Fig. 3. Magilla Verification Process

4.1 A sample Magilla program

In this section we will present a simplified sample Magilla program for IPv4 forwarding. The program consists of five Magilla instructions: DISPATCH, ETHERNET, IP, IP CLASSIFY, and EMIT. Each Magilla instruction is written in a high level C-style function. Every Magilla program has a DISPATCH instruction as the first instruction. DISPATCH assumes that a packet header is already copied and accessible in scratch memory. The programmer manipulates the packet data by accessing the scratch memory which in this case is organized as two distinct functional units MEMX and MEMY. The process of translating this C-style program to equivalent Verilog is fully automated by our compiler.

The functional units are specified at the beginning of the program using Pragma directives. Each instruction in the program consists of a number of C-style function calls to functional units. At the end of each instruction we have a explicit jump switch which specifies the next instruction.

In order to validate the functionality of our router we compared its output, packet-by-packet, with the output of Click software router [8]. We used several equinix-chicago CAIDA [4] traffic dumps as the input traffic to both the Magilla and Click routers. In order to measure the performance, we extracted only minimum-sized packets from the CAIDA traffic traces and used them as the input to our verification process.

The figure 3 shows the verification process of Magilla.

```
#include "Packet.mag"
#include "ETHERNET.mag"
#include "IP.mag"

#pragma NP_functional_unit(MEMX)
#pragma NP_functional_unit(MEMY)
#pragma NP_functional_unit(LOOKUPX)
#pragma NP_functional_unit(LOOKUPY)

bit_vector PP[20];
bit_vector Da[32], Sa[32];
```

```
bit_vector Daport[8], Saport[8];
bit_vector Chksum[16];
bit_vector Packet_status[32];
bit_vector Rpd_status[8];
bit_vector Packet_size[16];

instr_addr_t  NP_INSTR_DISPATCH ()
{

    l2protocol_t l2protocol;
    //Rbb and Rbi are base and index
    //to packet receive buffer

    PP = MEMX.read(Rbb+Rbi);
    Packet_status = MEMY.read(Rbb+Rbi);

    //Extract layer 2 protocol, packet status,
    //and packet size
    l2protocol = Packet_status.L2PROTOCOL;
    Rpd_status  = Packet_status.RPD_STATUS;
    Packet_size = Packet_status.PACKET_SIZE;

    //Jump to appropriate instruction based
    //on the packet layer 2 protocol
    NP_switch (l2protocol) {
       case PPP:
           NP_INSTR_PPP;
       case ETHERNET:
           NP_INSTR_ETHERNET;
       default:
           NP_INSTR_EXCEPTION;
    }
}

instr_addr_t  NP_INSTR_ETHERNET ()
{
    l3protocol_t l3protocol;
    protocol_t wordy;

    //Extract layer 3 protocol
    wordy = MEMY.read(PP, ETHERNET_L3PROTOCOL_WORD);

    l3protocol = wordy.L3protocol;

    //Jump to appropriate instruction based
    //on the packet layer 3 protocol
    NP_switch (l3protocol) {
       case IP:
           NP_INSTR_IP_ADDR;
       default:
           NP_INSTR_EXCEPTION;
    }
}

instr_addr_t  NP_INSTR_IP ()
{
    protocol_t wordx;
    protocol_t wordy;

    //Extract destination and source IP
    //address
    wordx = MEMX.read(PP, ETHERNET_IP_DA_WORD);
    wordy = MEMY.read(PP, ETHERNET_IP_SA_WORD);
    Da = wordx.D_address;
    Sa = wordy.S_address;

    NP_switch () {
       default:
           NP_INSTR_IP_CLASSIFY;
    }
}

instr_addr_t  NP_INSTR_IP_CLASSIFY()
{

    protocol_t wordx;
    packet_metadata_t wordy;
    router_port_t Da,Sa;

    Daport = LOOKUPX.search(Da);
    Saport = LOOKUPY.search(Sa);

    wordx =  MEMX.read(PP, ETHERNET_IP_TTL_WORD);

    MEMY.write(Rbb+Rbi, STATUS_WORD) =
    Packet_status & RPB_AVAILABLE_MASK;

    //Extract the time to live and
    //checksum of the packet
    Ttl = wordx.TTL;
    Chksum = wordx.CHKSUM;

    Status =  OK;
    if (NOT_VALID_ADDRESS(Daport))
       Status =  INVALID_D_ADDRESS;

    if (NOT_VALID_ADDRESS(Saport))
       Status =  INVALID_S_ADDRESS;

    NP_switch (Status) {
       case OK:
           NP_INSTR_EMIT;
       default:
           NP_INSTR_EXCEPTION_ADDR;
    }
}

instr_addr_t  NP_INSTR_EMIT()
{
    protocol_t Ttlword;

    //Update the TTL and checksum
    Ttl = Ttl -1;
    Chksum = Chksum + 0x0100;
    Chksum = Chksum + 100;

    Ttlword =
        {Ttl, Ttlword[23:16], Chksum[7:0],
         Chksum[15:8]};

    //Fbb and Fbi are base and index
    //to packet forward buffer

    MEMX.write(PP, ETHERNET_IP_TTL_WORD) =  Ttlword;
    MEMY.write(Fbb+Fbi) =
        {FPB_TAKEN, Dport, Packet_status[23:0]};

    NP_switch (Status) {
       default:
           NP_INSTR_DISPATCH_ADDR;
    }
}
```

5 Conclusions and Future Work

In this paper, we describe our methodology for generating specialized hardware from an application written in a domain-specific language. We believe the methodology can be using in a wide range of application areas. We have a working compiler that accepts high-level applications as an input and produces synthesizable Verilog. We have written a network processing application and compiled it using that compiler. We are still in the process of generating experimental results. We have been able to generate a version of the network processor that achieves 100M packets/sec in simulation using roughly 85% of a Virtex 5 TX240T FPGA including the functional units. We selected that FPGA since it is the one used

on the latest NetFPGA board and is a knee-of-the-curve FPGA, rather than the biggest one available.

The packet I/O interface of that implementation does not comply with the pin budgets available in the selected FPGA, but only because we have not yet incorporated the Xilinx multi-gigabit transceiver (GTX) modules [14] that will provide the capability to move packets in from the framer and out to the traffic manager. Our results were generated by compiling the Magilla source code to Verilog using our compiler and then passing that Verilog through Xilinx synthesis and place-and-route (ISE 10.3) and using the place-and-route information to determine performance. We expect to run a subset of Magilla (we do yet not have access to an FPGA board with the appropriate interfaces) on a real FPGA platform. We then plan to use the same methodology to implement applications in other domains.

REFERENCES

[1] EZCHIP NP-4 network processor. http://www.ezchip.com/.

[2] ANTLR compiler compiler tool. http://www.antlr.org/.

[3] AutoESL. http://www.autoesl.com/.

[4] The Cooperative Association for Internet Data Analysis. http://www.caida.org/.

[5] CatapultC. http://www.mentor.com/.

[6] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: exploiting parallelism to scale software routers. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 15–28, New York, NY, USA, 2009. ACM.

[7] Intel®IXP2800 Network Processor Hardware Reference Manual. ftp://download.intel.com/design/network/manuals/27888201.pdf, August 2004.

[8] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000.

[9] Hoang Le, Weirong Jiang, and Viktor K. Prasanna. A SRAM-based Architecture for Trie-based IP Lookup Using FPGA. In *FCCM '08: Proceedings of the 2008 16th International Symposium on Field-Programmable Custom Computing Machines*, pages 33–42, Washington, DC, USA, 2008. IEEE Computer Society.

[10] John W. Lockwood, Nick McKeown, Greg Watson, Glen Gibb, Paul Hartke, Jad Naous, Ramanan Raghuraman, and Jianying Luo. NetFPGA - An Open Platform for Gigabit-rate Network Switching and Routing. In *IEEE International Conference on Microelectronic Systems Education (MSE'2007)*, June 2007.

[11] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, 2008.

[12] KyoungSoo Park Sangjin Han, Keon Jang and Sue Moon. PacketShader: a GPU-accelerated Software Router. In *in Proc. of ACM SIGCOMM 2010, Delhi, India*, 2010.

[13] Cisco Toaster-4 network processor. http://newsroom.cisco.com/dlls/partners/news/2004/pr_prod_06-09.html.

[14] Xilinx Virtex-5 TXT series FPGA. http://www.xilinx.com/publications/prod_mktg/pn2094.pdf.

[15] Xelerated HX network processor. http://www.xelerated.com/.