Automated Mapping from a Domain Specific Language to a Commercial Embedded Multiprocessor

ABSTRACT

Application specific programmable systems are capable of high performance implementations while remaining flexible enough to support a range of applications. Architects of these systems achieve high performance through domain specific optimizations, often introduced at the expense of programming productivity. We examine one of the most performance critical and time consuming steps to arriving at efficient implementations on these platforms: the mapping of an application to a target architecture. We accelerate this step by constructing a model of the architecture which captures its key features while being amenable to automated mapping. With an analogous representation of the application and mapping formulated as an *integer linear programming* (ILP) problem, we demonstrate this approach finds solutions that are *guaranteed* to be close to optimal solutions with respect to this model. These solutions enable efficient implementations of representative network applications on a commercial network processor family. We show this approach can produce an implementation comparable to a hand crafted design.

1. INTRODUCTION

Application specific programmable solutions represent the potential to service an application domain with high performance implementations, fast design times, and low per part costs. To deliver a flexible platform tuned to a particular application domain, vendors have developed *application specific multiprocessors* (ASMPs) for a variety of embedded markets including graphics, digital signal processing, and networking. To tap into their potential, programmers must be able to produce high quality software quickly.

A programmer of such a device typically begins the design process by describing the functionality of an application in a form ranging from an informal white board sketch to a functional model written in C. One natural way for a programmer to describe an application is with a *domain specific language* (DSL). Developed by domain experts, DSLs match their mental model of an application domain. DSLs enable high productivity by providing component libraries, communication and computation semantics, visualization tools, and test suites tailored to a given application domain. For networking, a popular DSL is Click [1], which is an actor oriented language customized for packet processing and packet flow visualization.

While capturing the functionality of an application can be done quickly, the job of implementing it on ASMPs is notoriously difficult. To reap the benefits of an ASMP, a designer must decide on a partitioning of his/her application and its mapping to processing elements and exposed memories. They must then write multiple, often multithreaded, sequential C programs according to this mapping, all the while dealing with special purpose hardware, contention, alignment issues, and race conditions. Further, simply arriving at a working implementation is not enough, as the reason for these devices existence is to deliver high performance solutions. In practice, performance pressure often drives designers to program at the lowest level, leading to long design times and to being locked in to a particular system mapping. The wide divide that separates the original abstract application description and the low level programming methods of the target, we call the *implementation gap*.

In the world of network processing, solutions addressing some of these problems have appeared [2] [3] [4] [5]. Higher levels of abstraction, more robust library elements, and fuller C language support are promoting reusability, portability, and better ease of use in general. However, as architects add more processing elements and exposed memories, the key problem to arriving at a high performance implementation is finding a good mapping of the application to the architecture, which is a problem that remains largely unsolved. Even with new programming environments, programmers must manually arrive at an assignment of those computational tasks to processing elements along with a layout of data in memories and intertask communication to physical communication links. These intertwined design decisions result in a large, irregular design space that has a large impact on the implementation's final performance. As vendors add more cores and more memory, this programming challenge is only becoming more difficult with each generation.

We tackle this time consuming, performance critical step for high performance multiprocessor architectures, by first constructing a model of the architecture that captures its salient features, while still being amenable to mapping to it auto-

matically. We build an analogous application model, which can be automatically generated from a DSL description of the application. For the resulting mapping problem, we formulate it as an *integer linear programming* (ILP) problem that simultaneously considers the assignment of tasks, data, and interconnect. We leverage a modern ILP solver (CPLEX [6] to solve it to a guaranteed bound within the optimal with respect to our models. To demonstrate our approach, we map the data plane of representative network applications onto the Intel IXP2xxx series, a line of high performance network processors. With no additional information from the designer, the tool flow produces an implementation that forwards at rates comparable to a hand crafted design.

The remainder of this paper is organized as follows: Section 2 reviews prior work. Section 3 covers the background of the DSL and target platform used for this work. Section 4 describes our solution. In Section 5, we show our preliminary results and Section 6 concludes and discusses future work.

2. RELATED WORK

In the field of networking, scheduling or mapping tasks to architectures has been examined before. Shangri-La [4] is an alternate flow from a DSL to the IXP2xxx series. Applications are partitioned and annotated with mapping information which is used by a runtime system to assign them to processing elements. Scheduling mechanisms have also been examined for a multiprocessor variant of Click [7], in which it was found that static assignment on a multiprocessor system can boost performance by increasing the amount of shared data locality on a given processor. The treatment of locating data in software exposed memories is of increasing relevance to programming community and it is examined in this work as well. A broad survey of techniques are covered by Panda, et al. [8]. For memory exposed application specific solutions, Zhuge et al. present algorithms that can be used for efficient variable partitioning on multiple memory DSPs [9].

ILP has proven to be a robust method to tackle mapping applications to embedded multiprocessors. Yang et al. have developed an ILP framework to minimize the use architectural resources considering computation allocation, targeting the IXP2400 [10]. Jin, et al. consider the mapping problem of tasks to a soft multiprocessor instantiated on an FPGA [11]. Eisenring, et al. present CoFrame, a modular and flexible framework for exploring architectural design spaces as well as mapping by using task graphs consisting of communication and computation [12]. Bender, et al. present an ILP formulation [13] which enables processing and communication cost trade offs. Wehmeyer, et al. also employ ILP in a uniprocessor setup to optimize for memory placement of instructions and data across multiple exposed memories [14].

Missing from these approaches is an integrated consideration of task allocation and data layout onto multithreaded processing elements with a diverse memory hierarchy. This is the key design problem when traversing the implementation gap for an application specific multiprocessor. Our work examines this problem in the context of a complete design flow for ASMPs, not an incremental improvement to an existing one. Domain specific knowledge contains useful assumptions to construct models which can be mapped quickly while still producing efficient implementations.

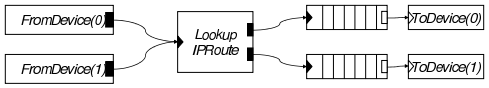

Figure 1: A Simple Click 2 Port Forwarder

3. BACKGROUND

In this section, we give some background about Click and the family of platforms we use as our demonstration vehicle – the Intel IXP2xxx series.

3.1 Click

Click [1] is an environment designed for describing networking applications. A Click description of an application is functionally complete, such that a programmer may test and refine his application using freely available tools on a general purpose platform. Based on a set of principles tailored for the networking community, Click has been used to described a variety of network applications [15] [16]. Applications are built by composing computational elements, which correspond to common networking operations like classification, route table lookup, and header verification. Prior work has shown that parallelism can be extracted and utilized for general purpose multiprocessor platforms [17]. Figure 1 shows a Click diagram of a simple 2 port packet forwarder, in which packets ingress through *FromDevice*, have their next hops determined by *LookupIPRoute*, are queued, and finally then egress through *ToDevice*.

3.2 Intel IXP2xxx

The IXP2xxx is Intel's most powerful family of network processors to date, capable of up to 23.1 billion instructions per second. Each member of the family has an XScale processor that is intended for control and management plane operations. There are multiple RISC processors with instruction sets tuned for packet processing called *microengines* (MEs). With instruction set architectures geared towards data plane processing, each microengine has hardware support for eight threads with a shared instruction store. To permit fast context swapping, each also has a hardware thread scheduler monitoring the readiness of each thread. To keep these cores busy, the memory architecture is divided into several regions: large off-chip DRAM, faster external SRAM, internal scratchpad, next neighbor registers, and local memories and register files for each microengine. Each region is under the direct control of the user, and there is no built-in cache structure for the microengines. Next neighbor registers allow producer-consumer relationships between neighboring microengines to avoid communicating through slower, globally shared memory. Shown in figure 2, the IXP2400 [18] is a midrange member of the IXP2xxx family with 8 microengines running up to 600 MHz, each with 8 hardware supported threads.

4. SOLUTION FRAMEWORK

A design flow starting with a Click to IXP2xxx flow is a good demonstrator of this crucial mapping problem. Previous work proposed a design flow for targeting such ASMPs [19], but considered only the task allocation aspect of the mapping problem for the simpler IXP1200. We constructed the design flow shown in Figure 3 for purposes of evaluating our solutions for the more general problem, which includes memory and communication. A designer starts by describing the application naturally in Click. This Click description

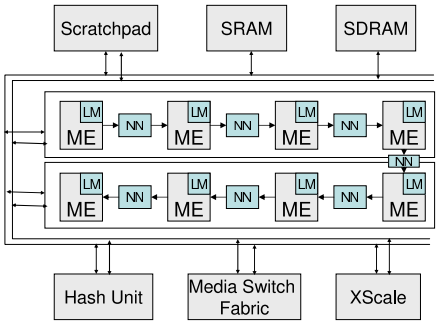

Figure 2: Block Diagram of the Intel IXP2400

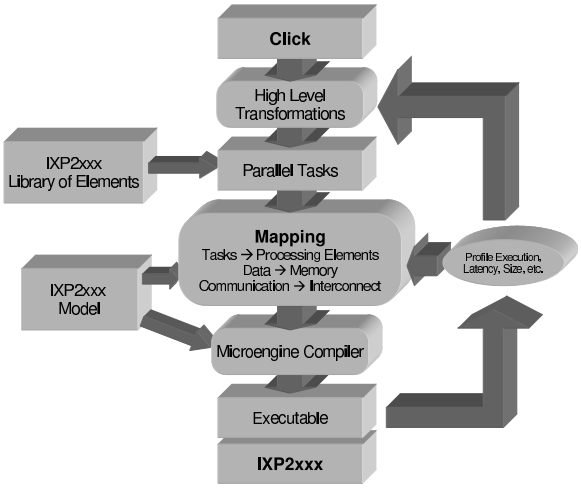

Figure 3: Automated flow from Click to the IXP2xxx series

is transformed into a task graph which contains target specific elements for computational tasks and data. We use NPClick [5] to supply our library of elements. Performance profiles of these elements along with a model of the target architecture are used by the mapping stage. The result of the mapping stage is a set of assignments which guides a Microengine C code generator. Intel's Microengine C compiler consumes these generated programs and produces the implementation that may then be run on the platform. Since prior stages use profiles of tasks, we allow for feedback to them. This paper focuses on the construction of an architectural model for this class of network processors that captures the salient features of the architecture, while still being amenable to automated mapping. Further, we develop an application model to match this architectural model that may be generated directly from a DSL.

4.1 Architectural Model

Based on our own experience with the embedded multiprocessors, we note that most performance critical mapping decisions are:

- allocating tasks to processing elements
- laying out data in memories
- assigning communication links to interconnect

As with traditional multiprocessor platforms, balancing the load of computation across processing elements has a high impact on performance. With memory exposed architectures such as the IXP2xxx series, the choice of where to locate a particular piece of data also has a large impact on performance. Locating a highly accessed piece of data in a faster, smaller memory can significantly improve performance. We capture these salient features of the architecture by modeling the platform as a directed graph in which the vertices represent processing elements and memories, and the edges represent available interconnect. Each processing element provides compute cycles while each memory has certain capacity for holding data and an average access time to write or read from it. To capture how computational blocks run on multithreaded cores, the model distinguishes between cycles used for execution and those spent with the processing element idle during long latency events such as memory accesses. For the IXP2xxx series, microengines are connected in a directed chain interleaved with next neighbor registers. Each local memory is connected solely to one microengine, while SRAM, DRAM, and scratchpad are connected to every microengine. This model may be parameterized for each of the members in IXP2xxx network processor family by specifying number of cores and average access time and capacity of memories.

4.2 Task Graph

A Click description of an application lacks implementation specific information necessary to effectively target this model. The application must be annotated and transformed into a intermediate representation, called a *task graph* that can be directly mapped to the architectural model. We define a task graph as a directed graph with nodes of computational blocks called *tasks*, T and the states which are read or written by them called *data*, D. These nodes are connected via edges that represent communication links with directionality indicating reads and/or writes between nodes. A task graph may be mapped onto an architectural graph by covering tasks with processing elements, data with memories, and links with interconnect.

To map to our model of multithreaded processing elements, we profile each task $t \in T$ for execution cycles consumed, e_t, and the idle cycles from latency events that cannot be altered by the mapping stage, known as fixed latency cycles, l_t. Tasks also experience latency accessing data assigned to memories by the mapping stage, called variable latency cycles, v_t. The completion time of a task is then $e_t + l_t + v_t$. We also characterized each datum $d \in D$. s_d denotes its size while $w^R_{t,d}$ and $w^W_{t,d}$ represent the read and write access weights respectively between a task and datum.

Hardware multithreading allows for fast thread swaps that enables processing elements to stay utilized during a long latency events by swapping out the waiting task and swapping in a ready one. To model task completion on a multithreaded core, we consider a processing element to be in one of two modes: (1) *compute bound*, in which latency events are effectively masked by useful execution done by other tasks and task completion is limited by the speed of execution, or (2) *latency bound*, in which there are not enough execution cycles to mask latency events, so task completion is limited by the speed of the longest task. To determine which of these two modes a given processing element is operating in, we calculate number of cycles needed to finish all the tasks on a processing element as the maximum of the sum of the execution cycles from all tasks assigned to that processing element and the total number of cycles needed to finish longest task, which is the sum of the execution cycles, fixed latency cycles, and the variable latency cycles.

Consider the mapping of one task connected to two memory elements and one special purpose unit, shown in Figure 4. An average execution of task t involves a few blocks of sequential instructions interleaved with memory and special purpose hardware accesses, which we then model as e_t, l_t, and v_t as shown below the execution trace. Note that v_t will change if either d_1 or d_2 is assigned to a memory with a different access time.

4.3 Mapping

To maximize for performance in the mapping step, we attempt to solve the following optimization problem: given task graph and an architectural graph as previously described, find a feasible mapping of tasks to processing elements, data to memories, and communication links to interconnect such that the maximum cycles needed by any processing element during an average period of tasks, called the *makespan*, is minimized. A summary of the constants used in this formulation are:

e_t	execution cycles consumed by task t
l_t	fixed latency cycles of task t
k_m	latency incurred by access memory m
s_d	space in memory consumed by data d in words
c_m	capacity of memory m in words
$w^R_{t,d}$	access weight of read from data d to task t
$w^W_{t,d}$	access weight of write from task t to data d
n	number of threads per processing element

Variables in the formulation are in the form of a selection matrix, in which a one represents that an architectural element is covering an application element. More formally they are:

$X_{t,p} \in \{0,1\}$	$\forall t \in T, \forall p \in P$ task t assigned to processor p
$Y_{d,m} \in \{0,1\}$	$\forall d \in D, \forall m \in M$ data d assigned to memory m
$Z^R_{t,p,d,m} \in \{0,1\}$	$\forall t \in T, \forall d \in D, \forall p \in P, \forall m \in M$ task t on processor p reads from data d in memory m
$Z^W_{t,p,d,m} \in \{0,1\}$	$\forall t \in T, \forall d \in D, \forall p \in P, \forall m \in M$ task t on processor p writes to data d in memory m
$E_{cycle} \geq 0$	max execution time over all processors (i.e. makespan)

These variables represent the universe of all possible mappings allowed by our model. To restrict this to the space of feasible solutions, we use the following constraints.

$$\sum_{p \in P} X_{t,p} = 1 \quad \forall t \in T \tag{1}$$

$$\sum_{m \in M} Y_{d,m} = 1 \quad \forall d \in D \tag{2}$$

$$\sum_{t \in T} X_{t,p} \leq n \quad \forall p \in P \tag{3}$$

$$\sum_{d \in D} s_d \cdot Y_{d,m} \leq c_m \quad \forall m \in M \tag{4}$$

$$\sum_{t \in T} e_t \cdot X_{t,p} \leq E_{cycle} \quad \forall p \in P \tag{5}$$

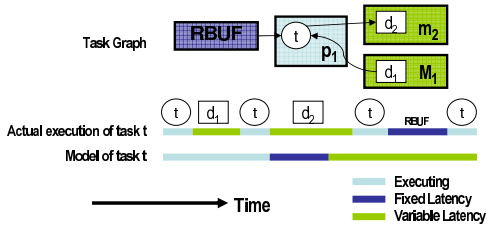

Figure 4: Multithreaded processing element model

$$(e_t + l_t) \cdot X_{t,p} + \sum_{m \in M} \sum_{d \in D} (w^R_{d,t} Z^R_{t,p,d,m} + w^W_{t,d} Z^W_{t,p,d,m}) k_m \leq E_{cycle} \quad \forall p \in P, \quad \forall t \in T \tag{6}$$

Constraints (1) and (2) ensures that each task and datum are covered by exactly one processor or memory, respectively. There is a corresponding covering constraint for links, which is omitted here for space. To utilize the hardware thread scheduler for swapping between tasks, our model ensures that each task may have its own thread as enforced by constraint (3). Capacity constraints for each memory is described by constraint (4). The makespan by definition must be greater than or equal to the number of cycles consumed on each microengine in a given period, which is enforced by constraint (5). Constraint (6) forces the makespan to be greater than the completion time of any task. Note that the summation in this constraint captures the variable latency cycles, v_t, for task t. After we are constrained to feasible mappings, we minimize for makespan, E_{cycle}.

4.4 IXP2xxx Topology Constraints

A single chip multiprocessor has a fixed topology. Any feasible mapping should have the application's communication links covered by architectural communication resources. The following constraints describe the restrictions imposed by the topology of the IXP2xxx series. Microengine number j is denoted by ME_j, while its corresponding local memory is LM_j. The next neighbor register that Microengine j reads from is represented as NN_{j-1} while it writes to NN_j.

$$Z^R_{t,\text{ME}_j,d,\text{LM}_k} = 0 \quad \forall d \in D, \forall t \in T, \quad \forall j \in \{1..|P|\}, \forall k \in \{1..|P|\} \text{ where } j \neq k \tag{7}$$

$$Z^W_{t,\text{ME}_j,d,\text{LM}_k} = 0 \quad \forall d \in D, \forall t \in T, \quad \forall j \in \{1..|P|\}, \forall k \in \{1..|P|\} \text{ where } j \neq k \tag{8}$$

$$Z^R_{t,\text{ME}_j,d,\text{NN}_k} = 0 \quad \forall d \in D, \forall t \in T, \quad \forall j \in \{2..|P|\}, \forall k \in \{1..|P|-1\} \text{ where } j \neq k+1 \tag{9}$$

$$Z^W_{t,\text{ME}_j,d,\text{NN}_k} = 0 \quad \forall d \in D, \forall t \in T, \quad \forall j \in \{1..|P|-1\}, \forall k \in \{1..|P|-1\} \text{ where } j \neq k \tag{10}$$

Constraints (7) and (8) force data linked to a particular task to not be associated with a local memory not on the same microengine. Constraints (9) and (10) ensures that data being used as a producer-consumer link may only be assigned to a next neighbor register if the associated tasks exist on microengines before and after the register, respectively.

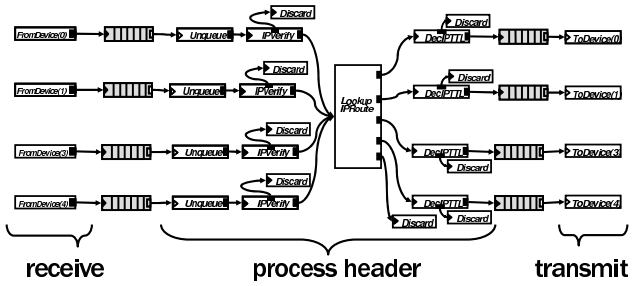

Figure 5: Click description of the dataplane of a 4 port IP forwarder

These constraints capture the basic features of the IXP2xxx necessary to arrive at high performance implementations, but we acknowledge that certain applications will require guidance from a designer to arrive at efficient or even feasible mappings. Such guidance could include binding tasks to particular microengines or clustering tasks. As benefit of using ILP, this formulation can accommodate these or many other conditions like it without modification to the original constraints.

5. RESULTS

We implemented a set of representative applications using our design flow. We created a 10 port IPv4 forwarder for the IXP2800, which forwards at a rate of 5.5 Gbps on 64 byte packets. For the IXP2400, we have constructed the dataplane of a network address translator which rewrites and forwards packets from a private network to another as described in RFC3022 [20]. It forwards and rewrites at a rate of 1.6 Gbps on 64 byte packets. In these cases, the designer supplies a Click description of the application and the implementation is automatically generated. For a detailed example of our flow, we present a 4 port Gigabit Ethernet IP forwarder implemented on the IXP2400 which is directly comparable with other frameworks. The following section describes the IPv4 forwarder application and how it moves through our design flow.

5.1 Application Description

Figure 5 displays the Click description of the application pictorially. Packets move from left to right through the diagram. They ingress through *FromDevice* and are immediately queued. Packets are dequeued and continue through a header verification process, a next hop lookup based on a 4 level trie table, and then have their time-to-live counters decremented. Finally they are queued and then egress through *ToDevice*.

5.2 Implementation

The programmer described the IPv4 forwarder in a 145 line Click configuration file. This contained element parameters and connectivity along with annotations on arcs representing packet distribution. This file also indicated duplication of execution chains to expose even more parallelism to be exploited by the mapping engine. The corresponding Click elements were profiled on the IXP2400 for average execution and fixed latency cycles. To arrive at these cycle counts each element was placed on an unloaded microengine and given its worst case traffic load. For example, the lookup element was exercised with packets that caused every level in the trie table to be accessed.

Using the original Click graph, the programmer annotations,

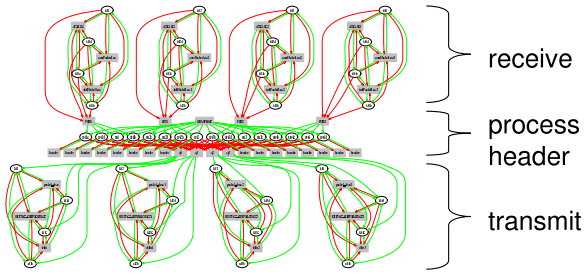

Figure 6: Task graph from Click description

and the profiled library elements for the IXP2400, the task graph shown in Figure 6 was generated automatically. Ovals are the tasks that correspond to execution chains in the Click graph. Boxes are data gleaned from the element library. Arcs represent read and write links between tasks and data. The tasks are annotated with the execution and latency based on the profile of the elements contained by them. The application has 42 tasks, 49 pieces of data, and many communication links.

The mapping formulation was applied to the task graph along with the model of the IXP2400 architecture. The mapping engine was run on a Sun Sparc workstation with 300MHz CPU with 256MB of memory. The solver could be configured to trade-off optimality guarantees with solve time. With an optimality guarantee of 15%, the mapping engine returned the solution shown in Figure 7 in 93 minutes. The white boxes enclosing ovals represent processing elements covering tasks and filled boxes encompassing gray rectangles as memories covering data. To reduce the makespan, the solver clusters tasks such that small, shared data is located in local memories, while queues that connect tasks on different processing elements are located in globally shared memory. The implementation is well load balanced with a minimal number of global memory access. To date, this performs as well as our highest performing hand mapped solution using this infrastructure.

To compare this framework versus existing ones, we tested this implementation on a variety of packet sizes in a cycle accurate simulator. We compared it to a hand-tuned Microblocks design as shown in Figure 8. We constructed within the same testbench an equivalent Microblocks design which produced results consistent with previously published performance numbers [10]. We were familiar with the architecture and we spent seven days constructing the Microblocks design, which included design, debug, and performance tuning. By contrast, the 145 line Click description took no more than a day to write and another hour and a half to arrive at an implementation. The Microblocks were able to out-perform our more portable infrastructure with more tightly optimized and architecture dependent code.

6. CONCLUSIONS AND FUTURE WORK

As application specific multiprocessors continue to add processing elements and expose distributed memory along with customized hardware features, programmers will continue to struggle to efficiently utilize the performance these devices. This work demonstrates that for these platforms, models can be written to capture the salient features of the architecture, and are still amenable to automated mapping to by powerful general purpose solvers. Further, they are flexible enough to accommodate designer guidance and new archi-

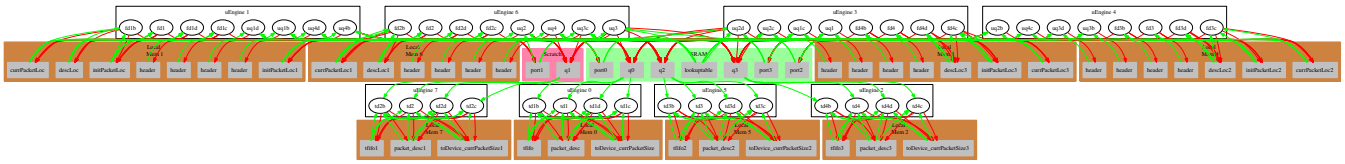

Figure 7: Automated Mapping Result

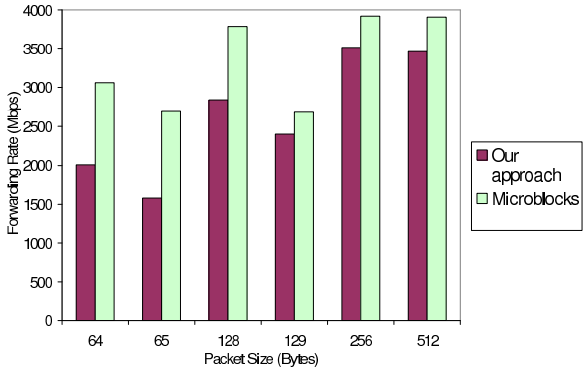

Figure 8: Forwarding Rates of Our Approach and Microblocks

tecture features. We have shown that a natural application description written in a domain specific language can be automatically transformed to an intermediate representation that matches this architectural model.

While we concede that for IPv4 forwarding, most designers will spend the extra time to use a less portable, harder to use design flow to arrive at a higher performing implementation. But considering designers have the benefit of application familiarity and reference designs, new complex applications will require automation support for designers. Considering mapping even the basic forwarding application can be a daunting task, more complex applications will prove impossible to implement in this framework. The approach presented here provides a new alternative by using a portable, natural domain specific language for design capture. Our design flow utilizing the mapping formulation described here to construct fast implementations. We have demonstrated this on a set of representative applications on powerful network processors by producing an implementation that is comparable a hand optimized design.

Many problems still remain for the effective deployment of complex, high performance multiprocessor systems including more application level optimizations, the ability to consider multiple implementations, and incorporating run time changes in workload. Also we believe the time needed to arrive at an efficient solution can be reduced through intelligent heuristics and tuning our general purpose solver to this problem. We plan to exercise this framework with other applications, as we believe that more complex and heterogeneous applications will further show the utility of this approach.

7. REFERENCES

[1] E. Kohler, et al., "The Click modular router," *ACM Transactions on Computer Systems*, vol. 18, pp. 263–297, August 2000.

[2] Teja, *Teja C: A C based programming language for multiprocessor architectures*, October 2003.

[3] R. Ennals, R. Sharp, and A. Mycroft, "Linear types for packet processing," tech. rep., University of Cambridge Computer Laboratory, 2004.

[4] H. Vin, et al., "A programming environment for packet-processing systems: Design considerations," in *Proceedings of Third Workshop on Network Processors and Applications (NP3)*, February 2004.

[5] N. Shah, et al., "NP-Click: A productive software development approach for network processors," *IEEE Micro*, vol. 24, pp. 45–54, September 2004.

[6] "ILOG CPLEX." http://www.ilog.com/products/cplex/.

[7] G. Calarco, C. Raffaelli, G. Schembra, and G. Tusa, "Comparative analysis of SMP Click scheduling techniques.," in *QoS-IP*, pp. 379–389, 2005.

[8] P. R. Panda, et al., "Data and memory optimization techniques for embedded systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 6, no. 2, pp. 149–206, 2001.

[9] Q. Zhuge, B. Xiao, E. Sha, and C. Chantrapornchai, "Efficient variable partitioning and scheduling for dsp processors with multiple memory modules," vol. 52, pp. 1090–1099, April 2004.

[10] L. Yang, et al., "Resouce mapping and scheduling for heterogeneous network processor systems," in *ANCS '05: Proceedings of Architectures for Networking and Communications systems*, pp. 19–27, ACM Press, 2005.

[11] Y. Jin, et al., "An automated exploration framework for FPGA-based soft multiprocessor systems," in *CODES-05*, pp. 273–278, September 2005.

[12] E. Michael, Z. Eckart, and T. Lothar, "CoFrame: A modular co-design framework for heterogeneous distributed systems," tech. rep., Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich, October 1999.

[13] A. Bender, "MILP based task mapping for heterogeneous multiprocessor systems," in *EURO-DAC '96/EURO-VHDL '96*, (Los Alamitos, CA, USA), pp. 190–197, IEEE Computer Society Press, 1996.

[14] L. Wehmeyer, U. Helmig, and P. Marwedel, "Compiler-optimized usage of partitioned memories," in *WMPI '04: Proceedings of the 3rd workshop on Memory performance issues*, (New York, NY, USA), pp. 114–120, ACM Press, 2004.

[15] E. Kohler, R. Morris, and M. Poletto, "Modular components for network address translation," in *Proceedings of OPENARCH '02*, pp. 39–50, June 2002.

[16] C. Sauer, M. Gries, and S. Sonntag, "Modular domain-specific implementation and exploration framework for embedded software platforms," in *DAC '05*, (New York, NY, USA), pp. 254–259, ACM Press, 2005.

[17] B. Chen and R. Morris, "Flexible control of parallelism in a multiprocessor pc router," in *Proceedings of USENIX '01*, pp. 333–346, June 2001.

[18] Intel, *Intel IXP2400 Network Processor Hardware Reference Manual*, November 2003.

[19] W. Plishker, K. Ravindran, N. Shah, and K. Keutzer, "Automated task allocation for network processors," in *Network System Design Conference Proceedings*, pp. 235–245, October 2004.

[20] P. Srisuresh and K. Egevang, "Traditional IP Network Address Translator (Traditional NAT)." RFC 3022 (Informational), jan 2001.