A Throughput-Driven Task Creation and Mapping for Network Processors

Lixia Liu[1], Xiao-Feng Li[1], Michael Chen[2], and Roy D.C. Ju[2]

[1] Intel China Research Center Ltd. Beijing, China
[2] Intel Corporation, Microprocessor Technology Lab Santa Clara, USA[⋆]

Abstract. Network processors are programmable devices that can process packets at a high speed. A network processor is typified by multi-threading and heterogeneous multiprocessing, which usually requires programmers to manually create multiple tasks and map these tasks onto different processing elements. This paper addresses the problem of automating task creation and mapping of network applications onto the underlying hardware to maximize their throughput. We propose a throughput cost model to guide the task creation and mapping with the objective of both minimizing the number of stages in the processing pipeline and maximizing the average throughput of the slowest task simultaneously. The average throughput is modeled by taking communication cost, computation cost, memory access latency and synchronization cost into account. We envision that programmers write small functions for network applications, such that we use grouping and duplication to construct tasks from the functions. The optimal solution of creating tasks from m functions and mapping them to n processors is an NP-hard problem. Therefore, we present a practical and efficient heuristic algorithm with an $O((n+m)m)$ complexity and show that the obtained solutions produce excellent performance for typical network applications. The entire framework has been implemented in the Open Research Compiler (ORC) adapted to compile network applications written in a domain-specific dataflow language. Experimental results show that the code produced by our compiler can achieve the 100% throughput on the OC-48 input line rate. OC-48 is a fiber optic connection that can handle a 2.488Gbps connection speeds, which is what our targeted hardware was designed for. We also demonstrate the importance of good creation and mapping choices on achieving high throughput. Furthermore, we show that reducing communication cost and efficient resource management are the most important factors for maximizing throughput on the Intel IXP network processors.

Keywords Network Processors, Throughput, Intel IXP, Dataflow Programming, Task Creation and Mapping

[⋆] Our current email addresses are: Lixia Liu(liulixia@purdue.edu), Xiao-Feng Li(xiao.feng.li@intel.com), Michael Chen(mrchen@gmail.com) and Roy D.C Ju(roy.ju@amd.com)

1 INTRODUCTION

While there are increasing demands for high throughput on network applications, network processors with their programmability and high processing rates have emerged to become important devices for packet processing applications in addition to ASICs (application-specific integrated circuits). Network processors typically incorporate multiple heterogeneous, multi-threaded cores, and programmers of network applications are often required to manually partition applications into tasks at design time and map the tasks onto different processing elements. However, most programmers find it challenging to produce efficient software for such a complex architecture. Because the type and number of processing elements that these tasks are mapped to greatly influence the overall performance, it is an important but tedious effort for programmers to carefully create and map tasks of applications to achieve a maximal throughput. It is also rather difficult to port these applications from one generation of an architecture to another while still achieving high performance.

The problem being addressed in this paper is the automatic task creation and mapping of packet processing applications on network processors while maximizing the throughput of these applications. We envision programmers writing small functions for modularity in our programming model, and focus on mapping coarse-grained task parallelism in this work. Hence, we apply grouping and duplication to construct tasks from functions as opposed to splitting functions to smaller tasks. Note that the cost model itself is applicable both for different task granularities and for function splitting. Our approach has been implemented in the Open Research Compiler (ORC)[4][6] and evaluated on Intel IXP2400 network processors. We also conjecture that our approach is applicable to other processor architectures that support multi-threading and chip multiprocessors (CMP).

The primary contributions of this paper are as follows. First, a throughput cost model is developed to model the critical factors that affect the throughput of a CMP system. Compared to other simpler models, we demonstrate our throughput cost model to be more accurate and effective. Second, we develop a practical and efficient heuristic algorithm guided by the throughput cost model to partition and map applications onto network processors automatically. This algorithm also manages hardware resources (e.g. processors and threads) and handles special hardware constraints (e.g. the limited size of control store allowed limits instructions for each task). Third, we have implemented and evaluated our partitioning and mapping approach on a real network processor system.

The rest of this paper is organized as follows. Section 2 introduces background on the Intel IXP network processors and the features of our domain-specific programming language, Baker. Section 3 states the problems of creation and mapping. Section 4 presents our throughput cost model and describes a practical heuristic algorithm for task creation and mapping. Section 5 evaluates the performance of three network applications using different heuristics in comparison to our proposed one. Section 6 covers the related work, and then we conclude this paper in Section 7.

2 BACKGROUND[3]

The Intel IXP network processors are designed for high-speed packet processing[2][3][4]. We briefly introduce the architecture of a representative one from Intel IXP network processors, Intel IXP2400. IXP2400 is composed of an Intel XScale core (XSC) for control plane processing, such as route table maintenance and system-level management functions, and eight 32-bit multi-threaded MicroEngines (MEs) for data plane processing that can process critical packet processing tasks at a high rate.

There are eight hardware thread contexts available in each ME, and they share the same execution pipeline. Each ME has a limited control store(4K instructions) to hold the program instructions that the ME executes. Intel IXP2400 has a four-level memory hierarchy: Local Memory, Scratchpad, SRAM and DRAM, which have capacities that increase proportionally to access latencies. However, the MEs have no hardware caches because of little temporal and spatial locality in network applications and the desire to avoid non-deterministic performance behavior. Each of the four memory levels is designed for different purposes. In particular, DRAM is used to hold the large packet data because of its fast direct communication channel to network interfaces and also because it has the largest capacity.

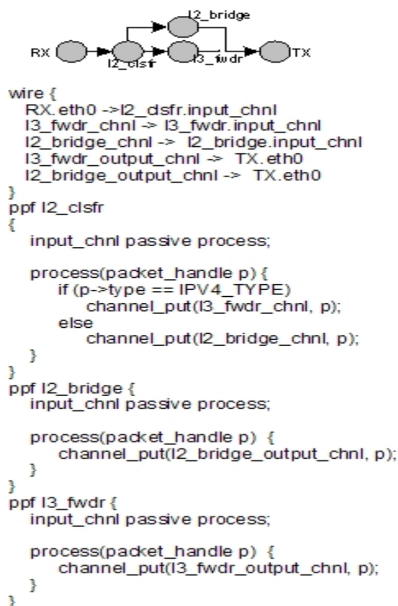

Fig. 1. A simple example of Baker application

There have been much research recently on programming the Intel IXP network processors[2][10][11][19][21]. Various programming languages have been proposed, and most of the evaluations discussed performance issues[10][11], e.g. NP-Click, a programming model for the Intel IXP1200, shows a IPv4 router written in NP-Click performs within 7% of a hand-coded version of the same application. We use the Baker[21] programming language in our study. Baker is a dataflow language designed specifically for network application development. Applications written in Baker are organized as a dataflow graph starting from a receiver (Rx) and ending at a transmitter (Tx). Actors in the graph are packet processing functions (PPFs), which are wired with communication channels.

Figure 1 shows the dataflow graph and code snippets of a simple IPv4 forwarding application. A PPF contains C-like code that performs actual packet processing tasks. A PPF typically consists of a few functions and is a logical processing stage in a whole application. Packets enter and exit a PPF through the two re-

[3] We present the essential background information here. Interested readers may refer to [3][21] for more details.

spective communication channel endpoints. A channel can be implemented in a efficient way, and it only carries a packet handle instead. PPFs are programmed with the assumption that they could run in parallel with other PPFs or with same copies running as multiple threads.

Baker also imposes several language restrictions to simplify code generation, e.g. recursion within a PPF is not permitted. Baker does not support recursion for two reasons: our survey of network applications indicates recursion is not required; and recursion would complicate and add overhead to runtime stacks on processors with heterogeneous memories.

3 PROBLEM STATEMENT

A network application can be represented as a directed data-flow graph where vertexes are composed of all of the PPFs as Figure 1 shows. An edge of the graph is defined as a channel that a PPF transfers packet data to another PPF. An IXP network processor includes multiple MEs and one XScale core. As stated earlier, each ME has a limited, fixed size of available control store. After processing, the resulting application is a list of tasks represented as T: $t_1 \dots t_p$. Each task is executed on one or multiple processors of IXP. If we want to map t_i to an ME, the number of instructions in a task t_i must be equal to or less than the size of ME's control store. Hence the mapping results are represented as (t_i, q_i) pairs, where t_i is a list of PPFs in the task and q_i is the set of processors on which t_i is executed. In our programming model, an ME can run only one task and a given task can be executed on one or multiple MEs. It is because much complexity of compiler backend is required to support multiple tasks on one ME, e.g. register allocation should split and allocate the registers of one ME according to different needs of those tasks. We can also extend our approach by considering threads in mapping strategy if multi-tasks are supported in the compiler backend.

We can apply various optimizations and actions to increase throughput. DRAM memory bandwidth is a precious resource on the IXP network processors due to the long latency in DRAM. We found that no more than two wide DRAM accesses (64B width) can be allocated to process each packet if we want to achieve the high throughput on IXP2400. Packets are stored in DRAM and the MEs have no cache, so we need to minimize the number of packet accesses for each processed packet. Unfortunately, we could incur additional DRAM accesses when communicating packets between pipelined tasks. For each task mapped to a processor, it must load packet header data from DRAM at its input channels and if the task modifies the packet header, it must write the changes back to DRAM before sending the packet to its output channels. The increased communication cost hinders the overall system throughput. Thus, we can reduce the communication cost by grouping two tasks into one and then running the new task on one processor.

As another fundamental property of pipelined tasks, if one task in a pipeline runs much more slowly than the others, the throughput of the pipelined tasks is determined by the progress of the slowest task. In this situation, this task can

be duplicated to run on more MEs and hence different packets can be processed in parallel by different MEs to improve the throughput. Both grouping and duplication play important roles to achieve high throughput in our algorithm. Therefore, we apply both grouping and duplication to address the automatic task creation and mapping problem.

4 OUR APPROACH: THROUGHPUT-DRIVEN PARTITIONING and MAPPING

We have developed a cost model to estimate system throughput and to guide the proposed partitioning and mapping algorithm. PPF acts as the smallest unit of application and the modularity of Baker provides an advantage for partitioning and mapping. As shown in Figure 2, the framework of our approach is composed of two major components: the Throughput Cost Model, and the Partitioning and Mapping Coordinator. The Partitioning and Mapping Coordinator iterates a number of times before settling on a final partition and mapping. The coordinator can choose to group or duplicate tasks. After selecting an action and updating the task list, the throughput cost model is queried to see whether the action benefits the throughput. If it is shown that the action does not benefit throughput, the coordinator will cancel the action, roll back all of the changes, and continue to the next iteration. We collectively call this a **Throughput-driven Partitioning and Mapping (TDPM)** approach.

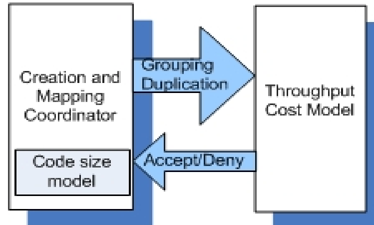

Fig. 2. The framework of our approach

The Partitioning and Mapping Coordinator also includes a code size model to estimate the number of instructions for each task to ensure that the fixed control store limit of the ME will not be exceeded. If the task has more instructions than the control store limit, it will abort when the task is loaded into the ME. The code size model estimates the number of static instructions from the intermediate representation of the application program. When two tasks are grouped together, the size of the newly created task must be recomputed. If the size of the newly created task is larger than the size of the ME's control store, the grouping action will be canceled and the changes will be rolled back.

In the rest of this section, we will illustrate how throughput will be affected by different actions using two examples and then describe our throughput cost

model in detail. We then show that obtaining an optimal partitioning and mapping solution is an NP-hard problem. Finally, we will present our polynomial-time algorithm that is both practical and effective for the problem that we are addressing.

4.1 Illustrating Examples

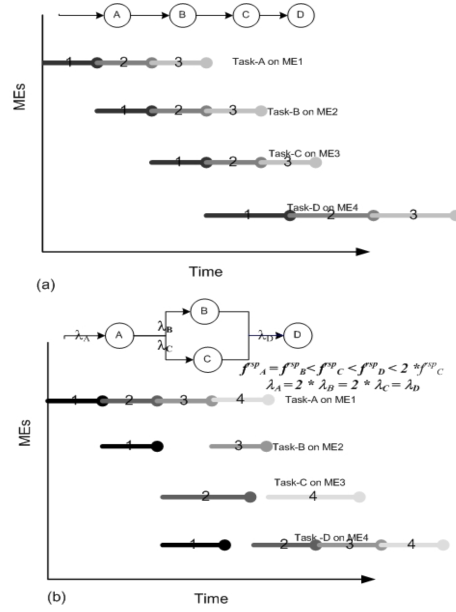

Fig. 3. Execution models for 4 tasks and 4 MEs on (a) linear chain of tasks and (b) non-linear chain of tasks

f^{rsp} is the average response time of each task in processing one packet, including both the execution and communication costs[4]. λ is the average input packet rate for a task. Without loss of generality, we illustrate throughput using two examples of four tasks running on four MEs in Figure 3. The length of each bar stands for the relative response time of the corresponding task. The number on a bar identifies the packet, and the same number refers to the same packet. In the linear chain of tasks in Figure 3(a), task-D has the longest response time f_D^{rsp}.Thus, the total system throughput is determined by task-D since it processes fewer packets than other tasks during the same period. The overall throughput can be approximated as the reciprocal of the maximum f^{rsp} ($\frac{1}{\max_{t_i \in T} f_i^{rsp}}$). There are many prior works that use similar models to estimate throughput for linear chains of tasks [13][14].

This framework does not hold for the applications that do not form a linear chain of tasks, like the one shown in Figure 3(b). In this dataflow graph of tasks, after completing task-A, half of packets go to task-B and the other half go to task-C. The input packet rates of different tasks are $\lambda_A = 2 \times \lambda_B = 2 \times \lambda_C = \lambda_D$. Let us assume that the response time of the four tasks has the relation of $f_A^{rsp} = f_B^{rsp} < f_D^{rsp} < f_C^{rsp} < 2 \times f_D^{rsp}$. Since task-C has the largest response time, the simple throughput model for the linear chain case will derive the overall throughput based on task-C. However, in this dataflow graph, the throughput should be determined by task-D, not task-C. This is because both ME2 and ME3 are partially idle during the interval between the arrivals of two successive packets, while ME4 is always busy. The utilization factor $f^{rsp} \times \lambda$ is introduced to depict how busy each processor is [17][18]. When a task has the largest utilization factor, the processor allocated to the task is the busiest. Such a processor dominates the system throughput because packets are waiting in the

[4] We use the term response time to differentiate with the execution cost. However, we notice that it has a different meaning from that of prior works.

processor's input queue. Therefore improving the overall throughput depends on minimizing the utilization factor instead of minimizing the response time of each task. This model also covers the cases of linear chain of tasks, because the utilization factor is proportional to response time when the input packet rates of all of the tasks are equal. In summary, we approximate the throughput of the dominant (or the slowest) task in our model as the reciprocal of the maximal utilization factor for all tasks on MEs, i.e. $\frac{1}{\max_{t_i \in T}(f_i^{rsp} \times \lambda_i)}$.

4.2 Throughput Cost Model

Suppose we have n available MEs in a given hardware configuration, with p the number of stages (normally the number of final tasks) in the processing pipeline. We approximate our system throughput (**H**) with the formula below:

$$H = k \times \lfloor \frac{n}{p} \rfloor \quad \text{if } n \geq p$$

k is the average throughput of the dominant (or the slowest) task in a given partition of the application, and it is $\frac{1}{\max_{t_i \in T}(f_i^{rsp} \times \lambda_i)}$. If n is larger than or equal to p, we can create $\lfloor \frac{n}{p} \rfloor$ copies of the packet processing pipeline so the throughput can be improved by $\lfloor \frac{n}{p} \rfloor$ times. When n is less than p, some tasks will be assigned to the XScale core and it requires a different throughput model. Since the throughput of the XScale core is rather low, we then apply grouping to reduce the number of tasks continually until the condition is met. Thus, this is not the case that this paper is focusing on. From this model, it should be clear that p needs to be minimized and k needs to be maximized at the same time to maximize the overall throughput. However, these two factors often impose conflicting requirements. If we try to reduce p, the PPFs may be grouped into a smaller number of tasks. However, this tends to increase the number of executed instructions and consequently the response time, of the slowest task, which reduces k. On the other hand, if we try to increase k, we can duplicate the slowest task or avoid grouping many PPFs into a given task. In either case, more MEs must be made available to hold all of the tasks, thus increases p. Therefore, a balance must be achieved to get the minimal p and maximal k that can result in the best system throughput.

p can be easily computed by tracking the number of tasks created. k is more complicated to compute. It must account for the effect of multi-threading on each ME, task duplication, and various kinds of costs associated with the response time of the slowest task. Thus, k depends on multiple factors: duplication factor(d), number of threads(γ), input packet rate(λ), and f^{rsp_core} , where f^{rsp_core} is the average response time of the task for each packet without duplication and with single-threaded execution on one processor.

Duplication of the slowest task can reduce λ in k linearly because packets can be processed by multiple MEs independently. Hence, k is proportional to the duplication factor(d) in our model. Multiple hardware threads on a given processing element also affect f^{rsp} in k because multi-threading can hide memory latency, communication cost, synchronization cost, etc. A precise model will

depend on the balance between computation and these costs. In our model, we approximate the benefit of multi-threading optimistically by scaling k with the value of γ.

After including the effect of duplication and multiple threads, k is modeled as: $k = \frac{\gamma}{\max_{t_i \in T}(f_i^{rsp_core} \times \lambda_i \div d_i)}$. The input packet arrival rate λ is computed from the input packet trace of the Profiler phase. The Profiler also provides the frequency of executing each node. The four remaining critical components are computation cost, communication cost, memory access latency and synchronization cost. f^{rsp_core} is modeled as a combination of these four critical components. The computation cost depends on frequency of each computation node and the latency for executing the instructions of the computation node. The memory access latency is computed from the memory access frequency, the size of data accesses, and the latency for accessing a specific memory level. The communication cost is derived from the invocation frequency, the amount of data transferred on each channel, and the communication latency. We calculate the communication cost by modeling the number of DRAM accesses since we need load and store packets from DRAM, although we have a fast runtime implementation of a channel to transfer packet handles between different processors. When we group two tasks into one, we reduce the communication cost between the two tasks since packets can typically be cached in a much faster way within the same processor. For the synchronization cost, we found that it depends both on the acquire, release and critical section costs of each lock and the number of threads involved. When tasks are duplicated, the synchronization cost must be recomputed because the number of threads involved increases.

4.3 Optimal Partitioning and Mapping

Using our throughput cost model, an optimal algorithm compares throughput for all of the grouping and duplication possibilities and derives the partitioning and mapping configuration with the best throughput. A large scale network application can be composed of a large number of PPFs. Without considering grouping, we can get the optimal mapping in $O(nm)$ time for m PPFs on multiple processors, which include n MEs and one XScale core. Each processor is assigned to the slowest task which can be computed in $O(m)$ time by getting the largest utilization factor among m tasks. There are $n+1$ processors, and if there is not enough MEs, we will assign the remaining tasks to the XScale core, so the optimal mapping without grouping is determined at $O(nm)$ time. Duplication is already considered in this case since one task can be assigned to multiple processors, which means that the task is duplicated multiple times. With grouping, the problem becomes more complicated and resembles a traditional clustering problem[22][23]. Just as a variation of a traditional bin packing problem, our problem attempts to maximize the throughput for each task and also packs maximal PPFs into each task to minimize the number of tasks. Thus, to find optimal partitioning and mapping is also an NP-hard problem because the traditional bin packing problem has been shown as NP-hard[24].

4.4 Heuristic Partitioning and Mapping Algorithm

Because finding an optimal partition and mapping of an application is NP-hard, we propose an iterative heuristic algorithm driven by our throughput cost model. This algorithm for the Partitioning and Mapping Coordinator is described in Figure 4.

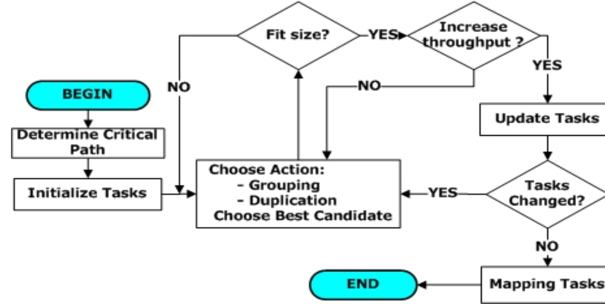

Fig. 4. The Framework of Practical Partitioning and Mapping Algorithm

This heuristic algorithm is primarily aimed at reducing communication cost. It first identifies the critical path of an application from statistics collected by a Profiler phase. Following that, we initialize new tasks with each PPF individually assigned to one task. Grouping and duplication for non-critical tasks is skipped because those tasks will simply be mapped to the XScale core. For all of the critical tasks, we then choose between duplication and grouping by analyzing the utilization factor of each task and the usage of hardware resources in the *Choose Action* step. We favor grouping except when task stages are imbalanced and there are available MEs for duplication. In the *Choose Best Candidate* step, we choose the best candidate according to the selected action. For grouping, the best candidate is chosen according to the benefit, e.g. the reduced communication cost between tasks. We build a candidate list prioritized by the reduced communication cost computed at initialization. The reduced communication cost can be estimated from channels that can be replaced by direct function calls. After partitioning, we are also required to ensure that there is no recursion within each task. Some channels replaced by calls may introduce recursion into the program, so we disallow those replacements in our algorithm. For duplication, the best processing candidate is the task with the largest utilization factor.

After choosing the action and a candidate, the candidate has to pass two additional checks before the action can be committed. The first check considers hardware constraints (e.g. whether the static instruction size of the task fits within the control store of an ME by the code size model). The second check evaluates the performance impact (i.e. whether the throughput increases or not). If either of two checks fails, we will abort the action and look for another action and candidate. This algorithm iterates until all tasks are examined. The

performance check in the algorithm simply uses the results of the throughput cost model. In the *Update Tasks* step, we update the task list and candidate list when the tasks are changed. In the end, we map the final partitioned tasks to the heterogeneous processors by running critical tasks on MEs and non-critical tasks on the XScale core. In this mapping step, tasks are sorted according to their computed utilization factors. We then map tasks to MEs and the XScale core in a descending order.

To partition and map m PPFs to n MEs and one XScale, the heuristic algorithm iterates at most $n+m$ times because the maximum possible groupings is m and the maximum times of duplications is n. We only spend $O(1)$ to choose the best candidate because the candidate list is ordered and the largest utilization factor is always tracked. The complexity of computing the candidate list during initialization is $O(m^2)$ since the candidate list can hold m^2 entries for every possible pair of tasks. We update the candidate list in $O(m)$ time because we change at most m entries. Hence, the total complexity for this algorithm is $O(m(n+m))$.

5 EXPERIMENTAL RESULTS AND EVALUATIONS

We have implemented our proposed throughput-driven partitioning and mapping approach in ORC based on our language, Baker. The experiments were conducted on an IXP2400 network processor, which has eight MEs and one XScale core. Two of the MEs are reserved for the packet receiver and transmitter respectively. Thus six MEs are available for each application.

We use three typical packet processing applications for evaluation. They are kernel applications for high-end network processors often used in network routers: **L3-switch**, **MPLS**, **Firewall**.

We experiment under different configurations by partitioning and mapping these three applications using different heuristics to evaluate our approach. The baseline configuration is marked as *Base*, which shows the worst-case scenario when all of the tasks are mapped to the XScale core. Other configurations are evaluated by taking additional factors, such as resource constraints, communication cost, and critical paths differentiation, into consideration. Configuration *RES* considers the control store limitation, hardware resources, and other costs except for communication cost. In this configuration, we never group any pair of critical tasks to reduce communication cost, but we can duplicate tasks and map them to MEs. Configuration *COMM* uses a greedy algorithm to reduce communication cost and performs simple resource management, e.g. duplicating all tasks equally. It also handles the control store limitation, but it does not consider other costs. Configuration *TDPM-S* is a simplified version from our throughput-driven partitioning and mapping approach. This version does everything in our approach except for differentiating between non-critical and critical paths. Configuration *TDPM-all* is our full approach. We evaluate the benchmarks using the 3 Gbps packet line rate with the minimum packet size of

64B in Figure 5. The X-axis shows the number of available MEs, and the Y-axis shows the forwarding rate, which is the key performance metric.

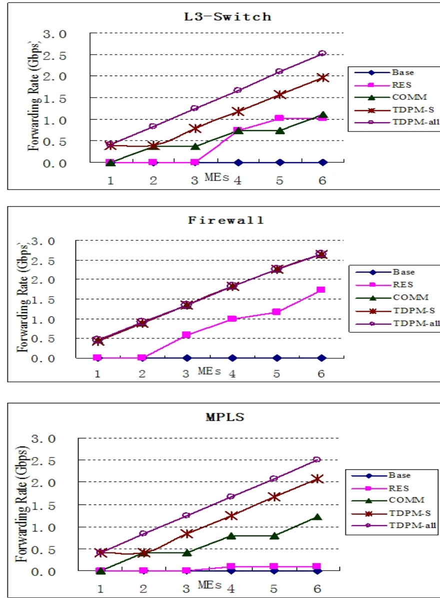

Fig. 5. Performance of three benchmarks under different configurations

In Figure 5, configuration *Base* receives a very low performance when all of the tasks are mapped to the XScale core. Configuration *RES* always performs better than the *Base* configuration. This is because configuration *RES* manages resource better than *Base* and can map the critical tasks to MEs. This shows the importance of resource management. For the *COMM* configuration, the curves for MPLS and L3-Switch are close to that of the *Base* configuration on a small number of MEs because critical tasks are mapped to the XScale core when the number of available MEs is smaller than what is needed. For MPLS, *COMM* typically has much better performance than *RES* because it reduces much of the communication cost that hinders the overall throughput. Certain compiler optimizations (e.g. packet caching and instruction scheduling) become more effective when the scope of a task increases (i.e. includes more instructions). However, the greedy method of *COMM* consistently achieves worse performance than our proposed approach *TDPM-all* for MPLS and L3-Switch. There are several reasons for the poorer performance. One reason is that configuration *COMM* considers only communication cost and does not obtain good load balancing, which causes inefficient resource usage. As such, it duplicates the faster tasks instead of slower tasks. Another reason is that *COMM* may combine non-critical code together with critical tasks, which could have adverse effects on certain traditional optimizations, e.g. increasing the register pressure and leading to more spill code. The Firewall benchmark is an exception. On this application, *COMM* shows little difference on performance compared to configuration *TDPM-S* and *TDPM-all* because all of the tasks of Firewall are critical.

Configuration *TDPM-all* performs better than configuration *TDPM-S* for MPLS and L3-Switch on larger numbers of MEs. This result shows the importance of managing hardware resource effectively by differentiating between non-critical and critical tasks and making more MEs available for critical tasks. For these three benchmarks, configuration *TDPM-all* achieves the same optimal partitioning and mapping as the optimal solution. In general, configuration *TDPM-all* provides the best performance and scalability for all three applications. It demonstrates that our throughput cost model is effective in modeling

key factors in throughput and that our partitioning and mapping algorithm produces efficient application tasks for the IXP network processors.

Table 1. Memory accesses for benchmarks

		Packet			Application		
		SCRATCH	SRAM	DRAM	SCRATCH	SRAM	Total
L3-Switch							
	TDPM-all	2	3	2	1	15	23
	TDPM-S	2	3	2	1	38	46
	COMM	2	3	2	5	299	311
	RES	8	3	6	1	19	37
Firewall							
	TDPM-all	2	10	1	3	17	33
	TDPM-S	2	10	1	3	17	33
	COMM	2	10	1	3	17	33
	RES	5	10	2	9	18	44
MPLS							
	TDPM-all	2	12	2	2	11	29
	TDPM-S	2	12	2	2	11	29
	COMM	3	12	2	2	11	30
	RES	4	12	8	2	13	39

To further understand where the performance benefit comes from, we measure the number of memory accesses for each packet in an application under major configurations in Table 1. We exclude configuration *Base* because the XScale core is not designed for fast packet processing, and hence memory accesses on the XScale core have much worse performance compared to MEs. From the results in Table 1, we can see that *COMM* can effectively reduce the number of DRAM accesses compared to *RES*, where most of the communication cost is attributed to the DRAM accesses mainly used to communicate packet information between tasks. *TDPM-S* and *TDPM-all* both reduce the number of SRAM accesses in application data, and the reduction is particularly effective on L3-Switch. However, in contrast to DRAM accesses, the reduction in SRAM accesses results in only a slight performance improvement.

We found that typically the best performing partition for our applications groups all of the critical code of an application into one task and leave as many MEs available for duplication as possible. The reason is that even with balanced pipelined tasks, the communication cost used to communicate among multiple tasks often has a large overhead from DRAM accesses, which degrades the overall throughput[5]. In Table 2, we show that we achieve the 100% forwarding rate for OC-48 with the minimum packet size of 64B on IXP2400 for all three benchmarks with our approach. OC-48 is the targeted best performance for all applications including hand-tuned code on IXP2400. We show the forwarding rates on different numbers of MEs used till the forwarding rate reaches 100%(only use four MEs among six available MEs). The performance is achieved by including not only the partitioning and mapping techniques presented here but also all other optimizations [1]. The good scalability (i.e. approximately a linear speedup) of

[5] It is a future work to investigate ways of using a less expensive communication mechanism than DRAM accesses, e.g. caching packet data in Next-Neighbor registers for an adjacent ME.

Table 2. Overall performances of benchmarks

MEs	L3-Switch	Firewall	MPLS
1	31.48%	30.79%	29.42%
2	62.66%	61.46%	58.82%
3	93.47%	91.87%	88.19%
4	100%	100%	100%

all applications shows that even in the presence of aggressive optimizations, our cost model and algorithm can effectively deliver high performance.

6 RELATED WORK

There is a significant body of prior work in partitioning and mapping for high throughput [13][14][15][16][19][20][22]. Choudhary et al. [15] addressed the problem of optimal processor assignment, assuming that no communication cost or communication cost can be folded into computation cost. Our experiments show that an effective model of communication cost is an important factor in an actual throughput model. Subhlok and Vondran [13][14] introduced methods to perform optimal mapping of multiple tasks onto multiple processors while taking communication cost into consideration. However, these works focus only on how to partition simple applications composed of linear chains of tasks onto multi-processors. The SMP platforms studied in these work have dramatic differences from IXP network processor. For example, in IXP network processors, there is no cache for the four levels of memory hierarchies.

The IXP-C compiler [16][19] and the Pac-Lang project [20] have also been exploring solutions of partitioning applications for network processors. In contrast to the parallel program that we use in our system, the IXP-C compiler assumes that users develop large sequential programs. Thus, the IXP-C compiler is responsible for splitting sequential programs into small tasks and then duplicating them to maximize throughput. The compiler enumerates different partitions to find a solution that can achieve the performance goal specified by users, but does not attempt to get the best overall throughput. We demonstrate a different approach to achieve high throughput on network processors. The Pac-Lang project currently expects the user to specify in a script file how the compiler should split, group, and duplicate an application. The primary advantage of this approach is that the program logic is clearly separated from how it is compiled for a specific architecture. However, this approach is not fully automatic. Significant knowledge of the processor architecture is required to write and tune a script file.

7 CONCLUSIONS

This paper has presented a throughput cost model, which can effectively model those key factors affecting throughput, and a polynomial-time algorithm guided

by the cost model to partition and map applications onto the Intel IXP network processors. The key idea in maximizing the overall throughput is to minimize the number of task stages in the application processing pipeline and maximize the average throughput of the slowest task at the same time. Our algorithm uses effective heuristics for grouping, duplication, and mapping to achieve superior performance while handling hardware constraints and managing hardware resources.

Compared to other simpler heuristics, the experimental results show that our approach is effective in achieving the best throughput. For all three applications, we were able to achieve the 100% packet forwarding rate for OC-48, which is the targeted performance for all applications including hand-tuned code on IXP2400. We also observe that reducing communication cost and conducting effective resource management are both important in achieving high throughput on the IXP network processors.

We conjecture that many of the partitioning and mapping techniques developed in this work are applicable to additional and more complex applications (such as streaming and multimedia applications) on other multi-core, multi-threaded processor architectures.

8 ACKNOWLEDGMENTS

We would like to thank all people who were involved in the project, especially Jason H. Lin, Raghunath Arun, Vinod Balakrishnan, Stephen Goglin, Erik Johnson, Aaron Kunze, and Jamie Jason at Intel Corporation, and the collaborators in the Institute of Computing Technology (ICT), CAS and University of Texas at Austin. We also appreciate the reviewers for their helpful feedback.

References

1. Michael K. Chen, Xiao-Feng Li, Ruiqi Lian, Jason H. Lin, Lixia Liu, Tao Liu and Roy Ju. Shangri-la: Achieving high performance from compiled network applications while enabling ease of programming. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'05)*, Chicago, IL, June 2005
2. Lal George and Matthias Blume. Taming the IXP Network Processor. In*Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, San Diego, CA, June 2003.
3. Intel Corporation. Intel IXP2400 Network Processors. *http://www. intel.com/design/network/products/npfamily/.*
4. Erik J. Johnson and Aaron Kunze. IXP2400/2800 Programming: The Complete Microengine Coding Guide. *Intel Press*, Hillsboro, OR, April 2003.
5. Roy Ju, Sun Chan, Chengyong Wu. Ruiqi Lian and Tony Tuo. Open Research Compiler for Itanium Processor Family. In *Proceedings of 34th Annual International Symposium on Microarchitecture (MICRO-34)*, Austin, TX, December 2001.
6. Roy Ju, Pen-Chung Yew, Ruiqi Lian, Lixia Liu, Tin-fook Ngai, Robert Cohn and Costin Iancu. Open Research Compiler (ORC): Proliferation of Technologies and

Tools. In *Proceedings of 36th Annual International Symposium on Microarchitecture (MICRO-36)*, San Diego, CA, December 2003
7. Network Processing Forum. IP Forwarding Application Level Benchmark. *http://www.npforum.org/techinfo/ipforwarding_bm.pdf.*
8. Network Processing Forum. MPLS Forwarding Application Level Benchmark and Annex. *http://www.npforum.org/techinfo/MPLSBench mark.pdf.*
9. E. Rosen, A. Viswanathan and R. Callon. RFC 3031 - Multiprotocol Label Switching Architecture. *IETF*, January 2001.
10. Niraj Shah, William Plishker and Kurt Keutzer. NP-Click: A Programming Model for the Intel IXP1200. In *2nd Workshop on Network Processors (NP-2)*, Anaheim, CA, February 2003.
11. Niraj Shah, William Plishker and Kurt Keutzer. Comparing Network Processor Programming Environments: A Case Study. In *2004 Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, HPCA-10, Madrid, Spain, February 2004.
12. Tammo Spalink, Scott Karlin, Larry Peterson and Yitzchak Gottlieb. Building a Robust Software-Based Router Using Network Processors. In *Proceedings of the 18th ACM symposium on Operation Systems Principles (SOSP'01)*, Banff, Canada, October 2001.
13. Jaspal Subholk and Gary Vondran. Optimal Latency-Throughput Tradeoffs for Data Parallel Pipelines. In *Proceedings of the 8th ACM symposium on Parallel Algorithms and Architectures (SPAA'96)*, Padua, Italy, 1996
14. Jaspal Subholk and Gary Vondran. Optimal Mapping of Sequences of Data Parallel Tasks. In *Proceedings of the 5th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPOPP'95)*, Santa Clara, USA, 1995
15. A.N. Choudhary, B. Narahari, D.M. Nicol, and R. Simha. Optimal Processor Assignment for a Class of Pipelined Computations. In *IEEE Transactions on Parallel and Distributed Systems*, April 1994
16. Long Li, Bo Huang, Jinquan Dai and Luddy Harrison. Automatic Multithreading and Multiprocessing of C Programs for IXP. In *Proceedings of the 10th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPOPP'05)*, June 2005
17. R.B. Copper. Introduction to Queueing Theory. Second Edition, New York: North Holland, 1981
18. Leonard Kleinrock, Queueing Systems Vol.1: Theory, *Wiley*, 1975
19. Jinquan Dai, Bo Huang, Long Li and Luddy Harrison. Automatically Partitioning Packet Processing Applications for Pipelined Architectures. In*Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'05)*, Chicago, IL, June 2005.
20. Robert Ennals, Richard Sharp and Alan Mycroft. Task Partitioning for Multi-Core Network Processors. In *Proceedings of International Conference on Compiler Construction (CC)*, 2005
21. Michael Chen, E.J. Johnson and Roy Ju. Tutorial: Compilation system for throughput-driven multi-core processors In *Proceedings of 37th Annual International Symposium on Microarchitecture (MICRO'37)*,Portland, OR, December 2004.
22. V. Sarkar. Partitioning and scheduling parallel programs for execution on multiprocessors. *MIT Press*, October 1989.
23. Apostolos Gerasoulis, Tao Yang. A Comparison of Clustering Heuristics for Scheduling DAGs on Multiprocessors. *J. Parallel Distrib. Comput.* 1992
24. Michael R. Garey and David S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness, *W.H. Freeman*. 1979