Approximation Algorithm for Process Mapping on Network Processor Architectures[1]

Chris Ostler, Karam S. Chatha, and Goran Konjevod
Department of Computer Science and Engineering, Arizona State University

Abstract— **The high performance requirements of networking applications has led to the advent of programmable network processor (NP) architectures that incorporate symmetric multi-processing, and block multi-threading. The paper presents an automated system-level design technique for process mapping on such architectures with an objective of maximizing the worst case throughput of the application. As this mapping must be done in the presence of resource (processors and code size) constraints, this is an NP-complete problem [1]. We present a polynomial time approximation algorithm guaranteed to generate solutions with throughput at least $\frac{1}{2}$ that of optimal solutions. The proposed algorithm was utilized to map realistic applications on the Intel IXP2400 (NP) architecture, and produced solutions within 78% of optimal.**

I. Introduction

Over the past decade communication technologies, especially the Internet, have experienced phenomenal growth. This development has been accompanied by an exponential increase in the bandwidth of traffic flowing through the various networks. Internet traffic has grown by a factor of four every year since 1997 (doubling every 6 months) [2]. This growth in traffic greatly outpaces the doubling of processor performance every 18 months as observed in Moores Law. The need for increased performance in traffic processing has led to the advent of programmable network processors that employ a variety of architectural techniques to accelerate packet processing including symmetric multi-processing (SMP), block multi-threading, and fast context switching.

Despite the architectural innovations that have been incorporated in current day NP, very little effort has been devoted towards making the processors easily programmable. Application development on NP requires the designer to manually divide the functionality among threads and processors, and determine the application mapping. This low-level approach to programming places a large burden on the developer, requiring a detailed understanding of the architecture. Consequently, the current situation leads to increased design time in the best case, and poor quality solutions in the worst case.

The paper addresses application development challenges on programmable multi-core NP architectures. In particular, we focus on SMP architectures that support block multi-threading. The Intel IXP series processors and AppliedMicro Circuits Corporation (AMCC) nP series processors are commercially available examples of such architectures that together dominate the market place with over 50% market share [3].

The paper presents an automated system-level design technique to overcome the application development challenges of SMP and block multi-threading based NP architectures. The technique takes as input an application and NP architecture specification, and obtains a mapping of the application on the target architecture such that the worst case throughput is maximized. The application is specified as concurrently executing processes that communicate through bounded first in, first out (FIFO) queues. The architectural specification includes the number of cores along with their code size constraints. We derive properties of optimal solutions, and present a strategy for generating high quality solutions. We propose a technique for solving the problem, and prove that the technique will generate solutions with throughput at least $\frac{1}{2}$ that of optimal solutions. The proposed technique is evaluated by experimenting with representative network processing applications on the Intel IXP 2400 NP architecture.

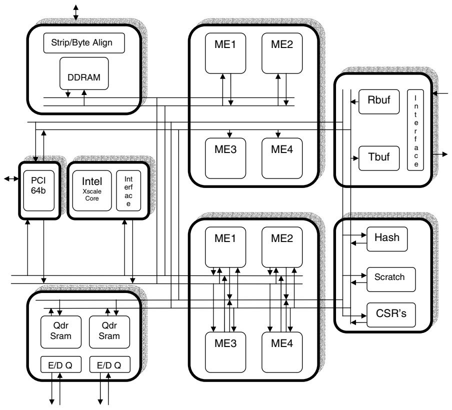


Fig. 1. IXP2400 Processor Architecture

The remainder of the paper is as follows: in Section II, we provide background information and a problem definition. We discuss previous work in Section III. Section IV details our proposed algorithm, and proves the quality bound on the solutions. In Section V, we present experimental results, and conclude in Section VI.

II. Background

A. SMP and Block Multi-threaded NPs

Let us consider the Intel IXP series of network processors. Fig. 1 shows the architecture of the IXP2400, which processes network traffic using eight micro-engines. Each micro-engine is a 32-bit RISC processor, which executes instructions from a local program memory of limited capacity. Further performance advantages (and design challenges) are presented by hardware supported block multi-threading (only one thread can execute on a micro-engine at a given time), and a diverse, heterogeneous memory architecture. Each micro-engine in the IXP2400 processor has hardware to maintain the context for up to eight threads, and allows for low cycle count context switching.

In data-driven applications such as network processing, where the same operations are performed on different packets,

[1] The research presented in this paper was supported by grants from the National Science Foundation (Career CCF-0546462 and CNS-0509540), and the Consortium for Embedded Systems.

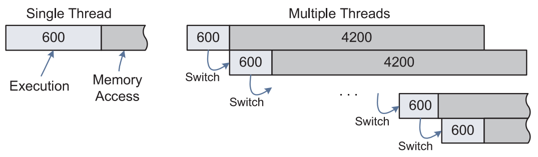


Fig. 2. Block multi-threading

block multi-threading allows data memory access times and related communication costs to be hidden. The left side of Fig. 2 denotes a single thread with an execution time of 600 clock cycles (cc), and data memory access costs of 4200 cc. The throughput in such a case is (1/4800) packets/cc. The right hand side depicts the scenario with 8 concurrent threads that perform the same operations on different packets. As the communication overhead of a single thread is hidden by the execution of 7 other threads, the overall throughput is (8/4800) = (1/600) packets/cc. Consequently, in block multi-threaded architectures the application mapping can be performed assuming that the communication costs will be amortized.

B. Application Description

We consider applications specified by a process network model. The processes are connected by FIFO queues, and communicate *only* using these queues. The contents of the FIFO queues may be actual data in the case of small data items (such as scalars), or pointers to memory in the case of large items (such as packets). We add an additional requirement that the process network be acyclic. We require that the processes are stateless; two executions of the same process can run concurrently and each produce the correct output without adverse effects.

We assume that a single execution of each process consumes a single data item from FIFOs leading in to the process, and produces a single data item on FIFOs leaving the process. In order to simplify the discussion, a set of data items consumed or produced by a single execution of a process will be considered a single item. Thus, the throughput of the system is determined by the number of data outputs produced per unit of time. As the processes are connected through finite FIFOs, the worst case throughput will be exactly that of the slowest throughput process in the network.

This application specification format is easily supported by SystemC and other system-level specification languages.

C. Design Flow

The design flow that we propose for mapping an application to a network processor architecture is shown in Fig. 3. Each process in the application is profiled against test streams to extract performance details which are used in mapping the processes to the available cores. These results represent the worst case scenario for the expected usage. Finally, the process mapping is used to generate the code that will be the implemented design.

As shown in the figure, we require a data mapping to be supplied in addition to specifying the functionality of the application. This remains a manual process to be completed by the designer. We do not address this portion of mapping the application; however, there have been techniques proposed for doing so [4]. Additionally, we will not examine the steps

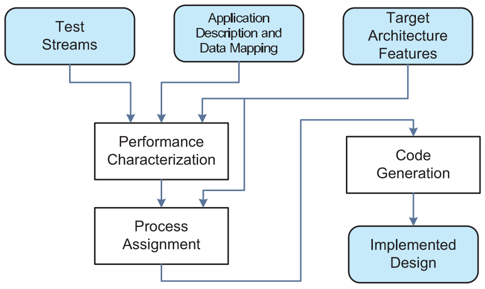

Test Streams
Application Description and Data Mapping
Target Architecture Features
Performance Characterization
Code Generation
Process Assignment
Implemented Design

Fig. 3. Design Flow

of code generation. Thus, our focus in this paper will be on assigning the processes in the application to the available processors.

D. Problem Description

For the sake of clarity, we will refer to a process in a process network as a job, and a processor as a machine. In these terms, the problem can be more formally defined as: given a set of jobs $\mathcal{J}$, and a number of symmetric machines M.

- Each job $j \in \mathcal{J}$ is characterized by the size of the code associated with the job (s_j) and execution time (t_j) on a machine.
- Each machine $m \in [1, M]$ has a limited amount of memory MEM_AVAIL in which to store programs; all jobs assigned to a machine must fit in the available memory

Our task is to generate a static assignment of jobs to machines so that the worst case throughput of the application is maximized.

III. Previous Work

In the past, researchers have explored the general problems of task allocation [5] and system-level design [6]. However, existing techniques do not address the issue of throughput maximization when mapping jobs to machines with limited program memory. NP-Click [7] is a programming model targeting network processors containing multiple processors, but it does not consider automated application mapping. Shah et al. [8] present a technique for task allocation, also for network processors. Further work was presented by Ramaswami, et al. [9], who used a randomized algorithm to assign task graphs to processing elements. However, none of the techniques presented in these works consider process transformation such as replication or merging which are essential (as we show later) for exploiting the full performance potential of SMP architectures.

Scheduling synchronous dataflow networks (and other application specification formats) is examined in many works, including [10] and [11], which build on the original work presented in [12]. The techniques primarily schedule task graph based application specification formats by utilizing heuristic techniques, such as list-based scheduling algorithms. However, they do not consider program memory limitations, and attempt to minimize application latency, rather than maximizing throughput. There have been approximation algorithms proposed for mapping applications to general multi-processor architectures [13], [14]. The algorithms presented, however, are not suitable for optimizing throughput.

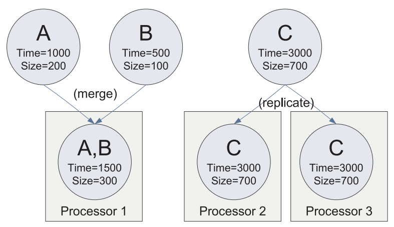


Fig. 4. Graph Transformations

This work overcomes the shortcomings of previous efforts by considering the limited program memory, and incorporates process transformations to better exploit SMP architectures.

IV. Job Assignment

There are two main factors that contribute to the complexity of assigning jobs to machines. The first is that of program memory limitations. Due to this constraint, not all combinations of job assignments are feasible; some combinations may require more program memory than is available. The second factor is that of parallelism. Running different jobs (or even multiple copies of the same job) concurrently on different machines can improve performance. However, it is difficult to determine how many instances of each job should be executed. Running too few instances may result in degraded performance due to unexploited parallelism; running too many may degrade performance by using processor cycles for computation that will not improve performance.

We outline a strategy for solving this problem. We first consider two transformations, merge and replicate, applicable to process networks. We then examine properties of optimal solutions, and establish theorems which are useful both for understanding optimal solutions to the problem, and also to direct and justify our solution strategy. We utilize the transformations through 'batching' jobs, and present an algorithm for solving the problem. Finally, we prove solutions will have throughput at least $\frac{1}{2}$ that of optimal.

A. Merge and Replicate Transformations

When applying the merge transformation, two jobs are combined into a new job, which will simply perform the computation of each contributing job. The contributing jobs are then replaced by the new job. Consider a process network with three jobs: A, B, and C. The left side of Fig. 4 shows the result of merging jobs A and B. Let us assume that we have three available machines, and an initial solution is that of assigning one job to each machine. The throughput of this solution is limited by the throughput of the slowest job - in this case, $\frac{1}{3000}$ for job C. Thus, merging jobs A and B does not change the throughput of the application, but frees a machine in the system.

To apply the replicate transformation, an additional instance of a job is created, and the throughput of that job is improved, as an additional iteration of the job can be completed in the same time period. Fig. 4(b) shows the result of replicating job C from the previous example. After completing this transformation, there are two iterations of job C which execute simultaneously. Therefore, the throughput of job C, and thus the application, is increased to $\frac{2}{3000} = \frac{1}{1500}$.

B. Optimal Solutions

Let us examine properties of optimal solutions to the problem. We will do so through the proof of two theorems. The first provides a definite upper bound on the throughput achievable for any application. The second describes one method in which a solution may be generated that will meet this theoretical upper bound. These two theorems will provide the basis for our solution strategy, as well as the means by which we will determine the approximation bound of our algorithm.

Theorem 1: An optimal solution will have a throughput no greater than

$$\frac{M}{\sum_{j\in\mathcal{J}} t_j}$$

Proof: The throughput of the application is determined by the number of times each job can be completed in a time unit. As every job must be completed once, there is a total of

$$\sum_{j\in\mathcal{J}} t_j$$

time units of execution required per completion of the application. Suppose that we are able to assign some fractional number of machines x_j to each job, such that

$$x_j = M \cdot \frac{t_j}{\sum_{j\in\mathcal{J}} t_j}$$

Assume there exists a solution S where this execution can be divided exactly evenly between each machine. Every machine will then run for

$$\frac{\sum_{j\in\mathcal{J}} t_j}{M}$$

time units, at which point all machines will simultaneously complete their assigned execution. During this time span, each machine has a utilization of exactly 100%. Thus, no solution can have greater throughput, and this provides an upper bound on the optimal throughput. ■

Theorem 2: When all jobs can be assigned to all machines without violating the code size constraints, such a solution is optimal.

Proof: Let us consider the throughput when all jobs are assigned to all machines. Each machine will complete a single instance of each job every $\sum_{j\in\mathcal{J}} t_j$ time units. As this will occur on every machine, then in this time span M instances of every job will be completed, yielding a throughput of

$$\frac{M}{\sum_{j\in\mathcal{J}} t_j}$$

As this is equal to upper bound on throughput shown in Theorem 1, this solution must be optimal. ■

From this we see that if the code size constraints are not limiting, it is trivial to find an optimal solution. However, if the code size can be exceeded, the problem of finding an optimal solution is NP-hard [1]. Thus, we will present an approximation algorithm for generating solutions.

C. Batching

Let us now consider a special case of the problem by introducing the concept of batching. Instead of assigning jobs to machines, let us instead assign jobs to some number of batches. Each job will be assigned to a single batch, and each

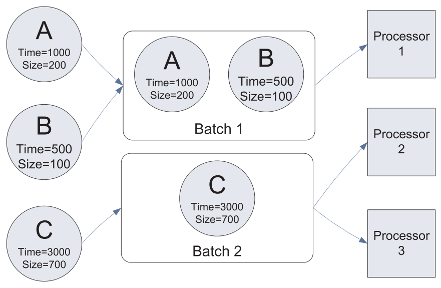


Fig. 5. A Batched Solution

machine will execute a single batch. Thus, the number of batches can never exceed the number of jobs, or the number of machines. If the former were to occur, there would be batches that do not contain jobs; if the latter occurred, there would be batches that are not assigned a machine. Thus for any set of batches $\mathcal{B}$, $|\mathcal{B}| \leq min(|\mathcal{J}|, M)$.

We can see that given a set of batches $\mathcal{B}$, we can determine the execution time t_b of each batch $b \in \mathcal{B}$ by summing the execution times of jobs assigned to the batch. The same can be done to determine the size s_b of the batch. Because machines will be assigned to execute an entire batch, all jobs in a batch must fit in the available program memory. Under these constraints, batching is equivalent to merging jobs until $|\mathcal{J}| = |\mathcal{B}|$, then replicating until $|\mathcal{J}| = M$. An example batched solution for the example process network from earlier is shown in Fig. 5.

We will now examine some properties of batched solutions. Again, we do so through the proofs of two theorems. The first concerns the execution of the jobs assigned to a batch, while the second shows that batching jobs does not inherently degrade the solution quality.

Theorem 3: In an optimal batched solution, each job in every batch will only be executed once.

Proof: Using the values of t_b and s_b described earlier, we will consider the batches to be jobs with the respective times and sizes. Let us assume we have an optimal solution S where every job in each batch is run once. From Theorem 1, the throughput of this solution is upper bounded by

$$\frac{M}{\sum_{j \in \mathcal{J}} t_j}$$

Now, suppose that there is a solution S^* where some job k is run an additional time. Since running a single job once more will not in itself allow for an additional completion of the application, the throughput of this solution has an upper bound of

$$\frac{M}{t_k + \sum_{j \in \mathcal{J}} t_j}$$

Obviously, this throughput is less than that of the first solution. Thus, in an optimal batched solution, each job in every batch will only be executed once. ■

This property is illustrated in Fig. 6. When a single job in the batch is executed more than once, the throughput is lowered. This continues until all jobs in the batch are executed an additional time, at which point the throughput returns to the original level, as it is equivalent to executing the original batch twice.

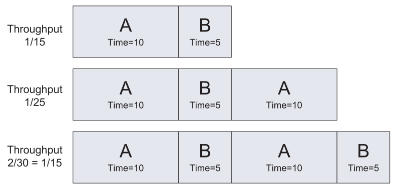


Fig. 6. Execution of Jobs Within a Batch

Theorem 4: The upper bound on throughput of a batched solution is the same as the upper bound on throughput for a non-batched solution.

Proof: If we again consider these batches as jobs as described earlier, using Theorem 1 we see that the throughput of the batched solution has an upper bound of

$$\frac{M}{\sum_{b \in \mathcal{B}} t_b}$$

Because of how the batches are defined, each job is assigned to a single batch. Thus,

$$\sum_{b \in \mathcal{B}} t_b = \sum_{j \in \mathcal{J}} t_j$$

so the upper bound on the throughput of the batched solution is equal to the upper bound on the throughput of any solution. ■

D. Mapping Algorithm

We now propose an algorithm for solving the problem as described, using a batching strategy. The ASSIGN_JOBS algorithm is shown in Fig. 7. The algorithm is based on the premise that given a batching of the jobs, it is possible to determine the optimal (possibly fractional) number of machines to be assigned to each batch. However, since it is not possible to assign machines fractionally, the values must be rounded to integer values. A simple scheme of rounding each value down to the nearest integer value would suffice, except for the case when the fractional number of machines to assign to a batch is less than 1. Since each batch must be assigned at least one machine, this value must be rounded up. This may not be possible in all situations (i.e. rounding all other values down may not free sufficient resources to round a value up), so the algorithm is called recursively to solve the problem for the remaining jobs and machines.

The algorithm first attempts to assign all jobs to all machines. If this is possible, it is an optimal solution, as shown in Theorem 2. If this is not possible, an attempt is made to solve the problem using 2 batches, then 3, until a sufficient number of batches is found. The reason for this is to ensure that one of the batches is constrained by the amount of available program memory. This is examined in more detail when showing the approximation bound.

1) Maximize Minimum Batch: The algorithm presented relies on the function MAX_MIN_TIME, which given a set of jobs and a number of batches will generate an assignment of jobs to batches such that the minimum execution time among the batches is maximized. We propose the following integer linear program that will solve this problem.

```
ASSIGN_JOBS($\mathcal{J}$, $M$)
1   if $\sum_{j\in\mathcal{J}} s_j \le$ MEM_AVAIL
2       assign all jobs to all machines
3   else
4       for $b$ := 2 to min($|\mathcal{J}|$, $M$)
5           $\mathcal{B}$ := MAX_MIN_TIME($b$, $\mathcal{J}$)
6           if no batching possible
7               continue
8           $\forall b \in \mathcal{B}$: $x_b = \frac{M \cdot t_b}{\sum_{l\in\mathcal{B}} t_l}$
9           $s$ := $b \in \mathcal{B} | x_b$ is minimum
10          if $x_s < 1$
11              assign 1 machine to $s$
12              ASSIGN_JOBS($\mathcal{J}/s$, $M - 1$)
13          else
14              $\forall b \in \mathcal{B}$: assign $\lfloor x_b \rfloor$ machines to $b$
15              if $\sum_{b\in\mathcal{B}} \lfloor x_b \rfloor < M$
16                  assign machines to minimal $b | x_b > 1$
17  return
```

Fig. 7. Job assignment algorithm

Problem Constants: We define the following constants:

- *Job Size:* Let s_j be the amount of program memory required by job j.
- *Job Time:* Let t_j be the time required to complete job j.

Variables: We use the following variables:

- *Job Assignment:* Let x_{jb} be 1 if job j is assigned to batch b, and 0 otherwise.

Constraints: We apply the following constraints to the problem:

- *Job Inclusion:* Every job must be assigned to a batch:
$$\forall j \in \mathcal{J} : \sum_{b\in\mathcal{B}} x_{jb} = 1$$
- *Program Memory:* All jobs assigned to a batch must fit in the available program memory:
$$\forall b \in \mathcal{B} : \sum_{j\in\mathcal{J}} x_{jb} \cdot s_j \le MEM\_AVAIL$$
- *Execution Time:* Determine the minimum execution time among all batches:
$$\forall b \in \mathcal{B} : t \le \sum_{j\in\mathcal{J}} x_{jb} \cdot t_j$$

Objective: As stated earlier, the objective is to maximize the minimum execution time among all batches; this is achieved by maximizing the value of t as the objective function.

E. Approximation Bound

We will now show that the ASSIGN_JOBS algorithm generates solutions with throughput at least $\frac{1}{2}$ that of the optimal solution. This is done through the proof of the following Lemmas. The first establishes a property of the batches generated by the MAX_MIN_TIME function. The others examine the assignment of machines to batches, and show that these assignments lead to solutions of guaranteed quality.

Lemma 1: In Step 11 of the ASSIGN_JOBS algorithm, the execution time of the minimum batch is limited by the available amount of program memory.

Proof: We first note that to reach this point in the algorithm, it is not possible to fit all the jobs on a single processor. Further, because of the structure of the algorithm, we know that $|\mathcal{B}| \le M$, and $x_s < 1$. Due to how the values x_b are calculated,
$$\sum_{b\in\mathcal{B}} x_b = M$$
Because $|\mathcal{B}| \le M$, then since $x_s < 1$, there must exist some batch $t \in \mathcal{B}$ where $x_t > 1$. Since t_s (and therefore x_s) is maximized within the available program memory, than it is not possible to generate a larger minimum batch. ■

Lemma 2: The assignment of a machine to s in Step 11 of the ASSIGN_JOBS algorithm is optimal.

Proof: Our proof will be by contradiction. Let us suppose we are given an optimal solution in Step 12, but that there is a solution to the original problem with better throughput. The only way to improve on the throughput of the optimally solved sub-problem is to move some of the computation from the sub-problem to the processor assigned to batch s. However, because batch s is limited by the size of the jobs assigned it, as shown in Lemma 1, this is impossible and therefore a contradiction. Thus, there cannot exist a solution with better throughput, so the assignment of a single machine to s is optimal. ■

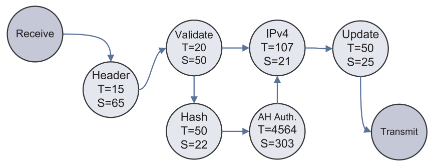
Lemma 3: The solution generated by the ASSIGN_JOBS algorithm has throughput of at least $\frac{1}{2}$ that of optimal.

Proof: There are three points in the ASSIGN_JOBS algorithm where machines are assigned to batches. The first (in Step 2) is reached only when all jobs can fit on a single machine. From Theorem 2, this is an optimal assignment. As shown in Lemma 2, the assignment done in Step 11 is also optimal. In Step 14, machines are assigned to batches based on the values of x_b. If the exact (fractional) value of x_b could be used, the resulting solution would be optimal. However, as only $\lfloor x_b \rfloor$ machines are assigned to each batch, the ratio of the actual to the ideal amount of time needed to complete the execution of a batch is $\frac{x_b}{\lfloor x_b \rfloor}$. This ratio approaches 2 as x_b approaches 2 from below. Since all batches will be completed in no more than twice the amount of time required in an ideal solution, the rounded solution will have a throughput of at least half that of the upper bound on the batched problem. Since from Theorem 4 this upper bound is the same as that of the general problem, ASSIGN_JOBS will generate solutions with throughput of at least $\frac{1}{2}$ that of the optimal solution. ■

V. Experimental Results

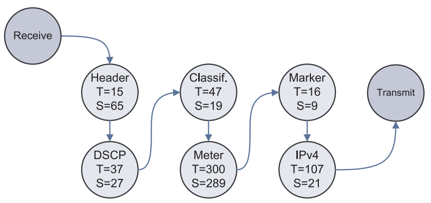
To validate the proposed ASSIGN_JOBS algorithm, we ran the algorithm against three sample applications commonly implemented on network processors. We compare the results of the solutions generated to the theoretical bound from Theorem 1. We also compare the results to those obtained by solving an ILP formulation for a batched assignment. It was not possible to compare the results with those obtained by solving an ILP formulation for a non-batched assignment, because of prohibitive runtimes required to solve such a formulation.

In mapping these applications, we targeted the IXP2400 architecture presented earlier. Although the IXP2400 has 8 processors, due to architecture constraints in how packets are received and transmitted, it was necessary to dedicate two processors, one each to the tasks of receiving and transmitting packets. Thus, we considered only the remaining six processors, running at 600 MHz. As discussed earlier, without code memory constraints, mapping jobs is a trivial problem. Thus, we artificially restricted the code memory to 400 words.



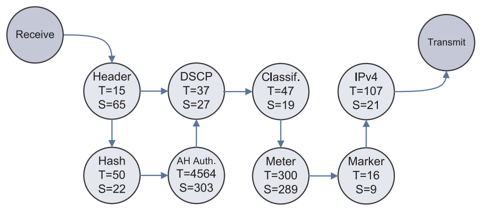
Process	Processor Assignment	
	ASSIGN_JOBS	ILP Form.
Header	1	1
Validate	1	1
Checksum	1	1
AH Authen.	2-6	2-6
IPv4	1	1
Update	1	1

Fig. 8. IPSec, IPv4



Process	Processor Assignment	
	ASSIGN_JOBS	ILP Form.
Header	1-3	1-2
DSCP	1-3	1-2
Classify	1-3	3-6
Meter	4-6	3-6
Marker	1-3	1-2
IPv4	1-3	1-2

Fig. 9. Diffserv, IPv4



Process	Processor Assignment	
	ASSIGN_JOBS	ILP Form.
Header	2-6	2-6
Checksum	1	1
DSCP	1	1
AH Authen.	2-6	2-6
Classify	1	1
Meter	1	1
Marker	1	1
IPv4	1	1

Fig. 10. IPSec, Diffserv, IPv4

Application	Upper Bound	Batched ILP		ASSIGN_JOBS	
		Throughput	Runtime	Throughput	Runtime
IPSec, IPv4	749.1 K/sec.	644.6 K/sec.	<1 sec.	644.6 K/sec.	<1 sec.
Diffserv, IPv4	6,897 K/sec.	6,857 K/sec.	8 sec.	6,000 K/sec.	<1 sec.
IPSec, Diff., IPv4	684.9 K/sec.	642.5 K/sec	4 sec.	642.5 K/sec.	<1 sec.

TABLE I: THROUGHPUT COMPARISONS

As the algorithm requires as input the process graph, annotated with process sizes and runtimes, we first generated these details. An implementation of the process network was profiled against a sample stream of packets, using a simulator supplied with the Intel IXA SDK. From the simulation we were able to obtain the necessary runtimes and sizes for each process, and annotated the graphs.

The three applications examined performed various combinations of AH authentication (part of IPSec), Diffserv Quality of Service (QoS), and IP version 4 lookup. While each application performed the IP lookup, the first also performed AH authentication, the second Diffserv QoS, and the third both AH authentication and Diffserv QoS. The process networks and the time (in cycles) and size (in words) of each of the jobs in the applications are shown in Figures 8, 9, and 10, respectively.

The mappings resulting from running the ASSIGN_JOBS algorithm and the ILP formulations are shown in the tables in Figures 8, 9, and 10. For two of the applications (IPSec, IPv4 and IPSec, Diffserv, IPv4), the ASSIGN_JOBS algorithm and the ILP formulation generated identical solutions. In the case of the Diffserv, IPv4 application, the solutions generated by the two methods differed.

The throughputs obtained by the ASSIGN_JOBS algorithm are compared to the upper bound and the throughput of the solutions generated by the ILP in Table I. The solutions were generated using dual Intel Xeon 2.8 Ghz processors, with 4 GB of memory. In these tables, throughput values are in terms of application completions per second, and are calculated by dividing the number of cycles between completion of the slowest process in the solution by the clock speed of the processors.

As can be seen from the results, the solutions generated by the ASSIGN_JOBS algorithm had throughput of no less than 78.4% of the theoretical upper bound, and 86.9% of the batched ILP. Although for one of the examples the ASSIGN_JOBS algorithm was unable to match the throughput of the solution generated by the ILP, it is also evident that the runtime of the ILP was substantially longer, even for very small examples. With larger applications, this gap would widen considerably, such that it would not be feasible to use the ILP to solve the problem.

VI. CONCLUSION

Tools and methodologies are essential to fully utilize the computational capacity of current day multi-core network processors. We have examined the problem of scheduling an application on network processors in order to maximize throughput. We proposed an algorithm to generate solutions to this problem, and showed that the algorithm generates solutions with throughput of at least $\frac{1}{2}$ that of an optimal solution. We compared solutions generated by the algorithm with the theoretical bound, as well as solutions generated by an ILP formulation, showing that the algorithm produced high quality results in feasible runtimes.

REFERENCES

[1] G. Ausiello et al. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer-Verlag, 1999.

[2] L. G. Roberts. Beyond moore's law: Internet growth trends. *IEEE Computer*, pages 117–119, 2000.

[3] B. Wheeler et al. Npu market sees broad-based expansion. *http://www.linleygroup.com/npu/Newsletter/wire050420.html*, Apr 2005.

[4] V. Ramamurthi et al. System level methodology for programming cmp based multi-threaded network processor architectures. In *Int. Symp. on VLSI*, 2005.

[5] B. Shirazi et al, editor. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, 1995.

[6] G. De Micheli et al, editor. *Readings in Hardware/Software Co-Design*. Kluwer Academic Publishers, 2002.

[7] N. Shah et al. Np-click: A programming model for the intel ixp1200. In *Workshop on Network Processors at the Int. Symp. on High Performance Computer Architecture*, 2003.

[8] N. Shah and K. Keutzer. Network processors: Origin of species. In *Int. Symp. on Computer and Information Sciences*, 2002.

[9] R. Ramaswamy et al. Application analysis and resource mapping for heterogeneous network processor architectures. In *Network Processor Design: Issues and Practices*. Morgan Kaufmann Publishers, 2005.

[10] A. Jantsch. *Models of embedded computation*. Morgan Kaufmann Publishers, 2005.

[11] S. Sriram and S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., 2000.

[12] E. Lee and D. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1), 1987.

[13] C. Chekuri. Approximation algorithms for scheduling problems. *Technical Report CS-TR-98-116, Stanford University*, pages 238–249, 2000.

[14] H. Shachnai and T. Tamir. Polynomial time approximation schemes for class-constrained packing problems. *Journal of Scheduling*, 4, 2001.