An ILP Formulation for System-Level Application Mapping on Network Processor Architectures[1]

Chris Ostler and Karam S. Chatha,
{chrisost, kchatha}@asu.edu
Department of CSE, Arizona State University, Tempe, AZ 85287.

Abstract

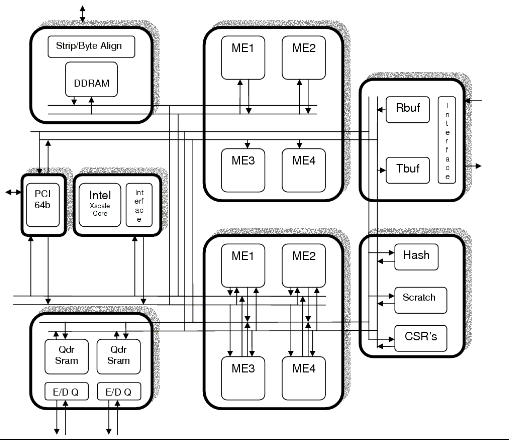
Current day network processors incorporate several architectural features including symmetric multi-processing (SMP), block multi-threading, and multiple memory elements to support the high performance requirements of networking applications. We present an automated system-level design technique for application development on such architectures. The technique incorporates process transformations and block multi-threading aware data mapping to maximize the worst case throughput of the application. We propose integer linear programming formulations for process allocation and data mapping on SMP and block multi-threading based network processors. The paper presents experimental results that evaluate the technique by implementing representative network processing applications on the Intel IXP 2400 architecture. The results demonstrate that our technique is able to generate high-quality mappings of realistic applications on the target architecture within a short time.

1. Introduction

Over the past decade data and voice communication technologies and the Internet have experienced phenomenal growth. These developments have 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) [1]. This growth in traffic greatly outpaces the doubling of processor performance every 18 months as observed in Moore's Law. The need for increased performance in traffic processing, and flexibility for end user customization to accommodate diverse applications has led to the advent of programmable network processors (NP) [2]. These processors employ a variety of architectural techniques to accelerate packet processing including symmetric multi-processing (SMP), block multi-threading, fast context switching, special-purpose hardware coprocessors, and multiple memories [2].

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 data mapping on the memory elements. This low-level approach to programming places a large burden on the developer, requiring a detailed understanding of the architecture and manual trade-off analysis between different design alternatives. Consequently, the current situation leads to increased design time in the best case, and poor quality designs 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 Applied Micro Circuits Corporation nP series processors are commercially available examples of such architectures that together dominate the market place with over 50% market share [3]. We present an automated, system-level technique that 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. We discuss optimization strategies, and present a system-level design methodology and integer linear programming formulations incorporating the optimizations. The technique and optimizations are evaluated by experimenting with representative network processing applications, targeting the Intel IXP 2400 NP architecture.



Memory Element	Type	Capacity	Access Time (cycles)	Context Switch
Data Mem.	Local	2560 B	3	No
Scratchpad	Shared	16 KB	60	Yes
SRAM	Shared	$\leq$ 256 MB	90	Yes
DRAM	Shared	$\leq$ 2 GB	120	Yes

Figure 1. Intel IXP 2400 network processor

The remainder of this paper is organized as follows: Section 2 gives the necessary background and defines the problem, we present previous work in Section 3, Section 4 describes the overall design methodology, Section 5 presents the linear programming based approach, Section 6 presents the experimental results, and finally Section 7 concludes the paper.

2. Background

2.1. Intel IXP2400 processor architecture

The Intel IXP 2400 architecture performs packet processing on eight 32-bit independent block multi-threaded micro-engines operating at 600 MHz, as shown in Figure 1 [4]. The micro-engines support either 4 or 8 threads. Each thread has its own register set, program counter, and controller specific local registers. Every micro-engine contains 4 KB of instruction store and 640 32-bit words of local data memory, divided equally among the threads on the micro-engine. The hardware supports block multi-threading; fast context switching allows a single thread to do computation while others wait for an I/O operation.

[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.

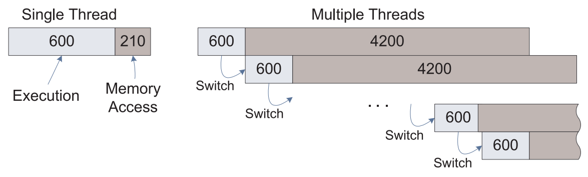


Figure 2. Optimization by multi-threading

Design	Mapping		Accesses		Total Overhead (cc)		Un-amortized Overhead (cc)		Perf. (pkts/
Threads	Local	Scratch	Local	Scratch	Local	Scratch	Local	S-pad	cc)
Single	*A&B*	-	70	0	210	0	210	0	$\frac{1}{810}$
Multiple	-	*A&B*	0	70	0	4200	0	0	$\frac{1}{600}$
Single	*A&B*	-	100	0	300	0	300	0	$\frac{1}{900}$
Multiple	-	*A&B*	0	100	0	6000	0	1800	$\frac{1}{825}$
Multiple	*A*	*B*	25	75	75	4500	75	0	$\frac{1}{675}$

Table 1. Multi-threading and data mapping

Each micro-engine also has access to three types of external memory: Scratchpad, SRAM, and DRAM, as shown in Figure 1. Context switches can be utilized to hide the access time for external data memories (but not for local data memories). The DRAM is used to buffer incoming packets as it has a fast interface (via DMA) with the media switch fabric that is responsible for receiving/transmitting the packets from the external environment.

2.2. Application specification

The application is specified as a network of concurrently executing processes that are communicating through finite sized (bounded) and blocking read/blocking write FIFOs, and blocks of abstract shared memory. The use of this model is not uncommon; as it is also supported by SystemC.

A block of abstract shared memory is used to store incoming packets. The packets are added and removed from the shared memory by the receive and transmit processes, respectively. In addition to packets, the abstract shared memory blocks can also include other data items such as arrays or tables that are shared among processes or are local to a particular process.

The individual FIFOs between two communicating processes store pointers to respective packet headers in the abstract shared memory. Each process (other than the source and sink) consumes one packet pointer, performs its respective computation and outputs the packet pointer to the out going FIFO. The overall process network can be represented as a directed acyclic graph.

2.3. Motivating examples

The worst case throughput of an application is governed by the throughput of the individual processes when they are mapped to the target architectures. Specifically, the entire application will be limited by the process with the lowest throughput. The following two examples explain the design tradeoffs when mapping an application to an architecture, and motivate our optimization strategies for maximizing the overall throughout of the application.

2.3.1. Multi-threading and data mapping

Let us consider the execution of a single process on a single processor. In trying to maximize the throughput of this process, we may either: minimize the time necessary to complete a single iteration of the thread; or use multi-threading so that the idle time caused by memory accesses is hidden by the execution of other threads. Take for example a process running on the IXP 2400 architecture that executes for 600 cycles to process a packet, and makes 25 memory accesses to a data structure *A* that requires 250B of memory, and additional 45 memory accesses to another data structure *B* which utilizes 750B of memory.

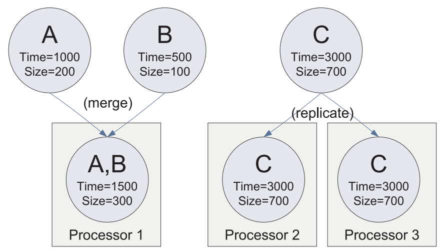


Figure 3. Process transformations

If we attempt to minimize the time to complete a single iteration, we will assign the data structures to the local memory, as it has the shortest access time. This introduces an additional 210 cycles (3 cycles for each of the 70 accesses) to the completion time of the process. As the local data memory accesses cannot be hidden by a context switch to a different thread, running multiple threads will not change the throughput of the process. The time between the output of consecutive packets will be 810 cycles (shown on the left hand side of Figure 2 and in the first row of Table 1).

Alternatively, we may attempt to maximize the throughput by utilizing multi-threading. On the IXP 2400 we can launch up to 8 identical threads. However, the local memory of the micro-engine and the registers are shared equally between the various threads. The total memory requirements for the 8 threads is 8KB. As the micro-engine local memory is 2.5 KB, they cannot be mapped to the local memory, and are mapped to the Scratchpad instead. Such a mapping introduces additional 4200 clock cycles (60 cycles for each of the 70 accesses) latency to each thread. This latency can be amortized by the execution of the 7 co-active threads, as shown in the right hand side of Figure 2 and the second row of Table 1. In this manner, a thread completes every 600 cycles, and thus, multi-threading leads to a 35% performance improvement.

The objective of multi-threading optimization is to hide the memory access latencies, which may not always be possible. Consider if data structure *B* is accessed 75 times. A multi-threaded implementation mapping both items to Scratchpad offers no performance advantage over a single threaded implementation, as shown in rows 3 and 4 of Table 1. However, if the data structure *A* is mapped on the local memory and data structure *B* is assigned to the Scratchpad, then multi-threading again gives a higher throughput (as shown in the last row of the table).

The benefit of multi-threading is strongly dependent upon the data mapping. Our technique successfully performs the design space exploration to select a data mapping that maximizes the benefit of multi-threading.

2.3.2. Process transformations

Let us now examine the effect of process transformations on the application throughput. Consider an application consisting of three processes, with execution time of 500, 1000, and 3000 cycles, which will be mapped to an architecture with three processors. As shown previously, communication time can be hidden using multi-threading, so we will not consider it at this point.

A simple mapping will assign each process to a processor. The overall throughput of the application is given by the slowest process and is equal to (1/3000). Alternatively, we can alter the process network by merging and replicating processes. The 500 cycle and 1000 cycle processes can be merged together into a single 1500 cycle process that sequentially performs the computation of the original processes. This reduces the number of processes to two. Mapping each of these processes to a single processor again results in the slowest process completing every 3000 cycles. We then replicate this process to the free processor, so that two instances execute in parallel. Thus, the slowest process will complete twice every 3000 cycles, a 100% performance improvement. This is illustrated in Figure 3.

Application of these transformations can lead to higher performance solutions. However, certain combinations of processes may require more code memory than is available, and therefore are not permissible. Our technique generates solutions which exploit the parallel processing supported on SMP systems.

2.4. Problem definition

2.4.1. Architecture features

Our methodology assumes a SMP based block multi-threaded architecture, specified by a tuple $\mathcal{A}\langle\mathcal{P},\mathcal{M}\rangle$. $\mathcal{P}$ denotes the set of processors. Each processor $p \in \mathcal{P}$ is given by another tuple $p\langle\kappa, C\rangle$ where $\kappa(p)$ is the set of numbers of threads supported by the processor (i.e. $\{4, 8\}$ for the IXP 2400), and $C(p)$ denotes the size of the code memory of the processor. Without loss of generality we assume that all processors have identical architectural features. For clarity, we use "memory" to refer to each of the data memories, and "code memory" to refer to the code memories of the processors. $\mathcal{M}$ denotes the set of memory elements that are available to the programmer. Each memory element $m \in \mathcal{M}$ is given by a tuple $m\langle C, \tau_a, L\rangle$ where $C(m)$ is the capacity of the memory, $\tau_a(m)$ is the time required to access the memory and $L(m)$ is a boolean that indicates whether the memory is local to a processor. A context switch can be performed when accessing non-local memory, but not when accessing local memory.

2.4.2. Application characteristics

The process model based application specification is profiled to obtain timing characteristics for each process and the access frequency for the various data items (which we discuss in Section 4). The profiled application specification is then captured in an intermediate format given by a tuple $\mathcal{N}\langle\mathcal{J},\mathcal{F},\mathcal{S}\rangle$.

$\mathcal{F}$ is the set of FIFOs. Each FIFO $f \in \mathcal{F}$ is represented by a tuple $f\langle j_p, j_c, \sigma\rangle$ where j_p is the producing process, j_c is the consuming process and σ is the size allocated for the FIFO. $\mathcal{S}$ is the set of shared memories. Each shared memory $s \in \mathcal{S}$ is denoted by a tuple $s\langle\sigma\rangle$ where σ is the size of the shared memory.

$\mathcal{J}$ is the set of processes. Each process $j \in \mathcal{J}$ is given by a tuple $j\langle\tau, S, \mathcal{D}_l, \mathcal{D}_f, \mathcal{D}_s\rangle$. $\tau(j)$ is the actual execution time of the process per packet (excluding memory accesses). This value has no dependence on the data mapping nor on the number of threads executed. S is the total code memory required by the process.

$\mathcal{D}_l(j)$, $\mathcal{D}_s(j)$, and $\mathcal{D}_f(j)$ are the sets of data items (local to the process), shared memories, and FIFOs accessed by the process, respectively. Each data item $d \in \mathcal{D}_l(j)$ is given by a tuple $d\langle\sigma, N\rangle$ where $\sigma(d)$ is the amount of memory required to store the item and $N(d)$ is the number of times the data element is accessed during the processing of a packet. Every data item $d \in \mathcal{D}(j)$ is used exclusively by a single thread of a single process. Data items shared between threads or processes will be captured in the set $\mathcal{D}_s(j)$. Each shared memory $d \in \mathcal{D}_s$ is given by a tuple $s\langle s, N\rangle$ where $s(d)$ is a shared memory $s \in \mathcal{S}$, to which d is assigned and $N(d)$ is the number of accesses. Each FIFO $d \in \mathcal{D}_f(j)$ is given by a tuple $d\langle f, N\rangle$ where $f(d)$ is a FIFO $f \in \mathcal{F}$ and $N(d)$ is the number of accesses made to the FIFO. As the application specification dictates that each process consume and produce exactly one token, $N(d)$ will always be 1 for FIFOs.

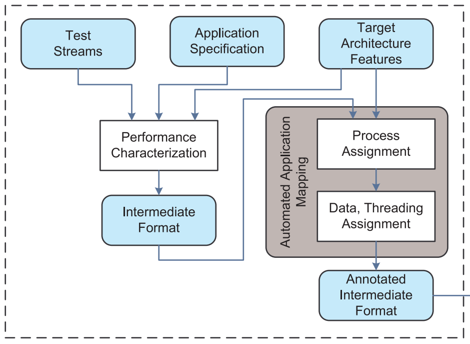

Test Streams
Application Specification
Target Architecture Features
Performance Characterization
Intermediate Format
Automated Application Mapping
Process Assignment
Data, Threading Assignment
Annotated Intermediate Format
Code Generation
Implemented Design

Figure 4. System-level design methodology

2.4.3. Objective

Given a characterized network processing application $\mathcal{N}$ and a target architecture $\mathcal{A}$, the objective is to obtain a mapping function $\mathcal{I} : \mathcal{N} \rightarrow \mathcal{A}$ such that the worst case throughput is maximized.

3. Previous work

Researchers have conducted design case studies [5–8] and proposed programming models [9–11] for multi-processor and block multi-threaded network processor architectures. However, they have not presented generalized methodologies for programming network processors, much less automated techniques for doing so. Commercial vendors [12] and academia [13, 14] have developed simulation based performance evaluators for network processors. Our automated techniques depend on application characteristics obtained by utilizing such a simulator to drive the optimizations.

Automated techniques for task allocation (including ILP based approaches [15, 16]) on multi-processor architectures are well researched [17, 18]. However, such generalized solutions are not suitable for networking applications and modern network processor architectures as they fail to consider process transformations and multi-threading and related data mapping effects, which are critical factors in determining the quality of the mapping on current day NP architectures.

Ramaswamy et al. [19] and Weng et al. [20] presented randomization algorithms for task allocation on network processors with an objective of minimizing latency (not throughput) without considering multi-threading or process transformations. Ramamurthi et al. [21] presented heuristic techniques for mapping applications to block multi-threaded multiprocessor architectures. Their techniques do not guarantee the quality of the solution. Plishker et al. [22] present an ILP formulation to map applications on to block multi-threaded multiprocessor architectures. However, they take into consideration only the code size as the constraint for merging various tasks. Also, as they do not consider replication of tasks, the throughput that can be obtained by the approach is severely limited. Further, the formulation primarily focuses on task allocation; they do not address data mapping and related optimizations that impact multi-threading.

4. Design Methodology

The system-level design methodology is shown in Figure 4.

4.1. Performance characterization

An initial mapping of the data items in the application on the memory elements is generated for profiling the application. For example on the IXP2400, the FIFOs and abstract shared memories are mapped to SRAM, and the local data items of each process

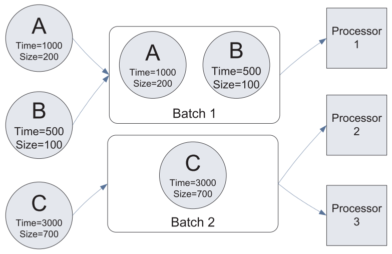


Figure 5. A Batched Solution

are mapped to the local memory of the micro-engine. Each of the processes is then profiled by utilizing simulation environments provided by the network processor vendors.

4.2. Application mapping

Our solution strategy consists of dividing the problem of mapping an application to an architecture into two stages, and solving each consecutively. The first stage assigns processes to processors, and the second stage performs multi-threading aware mapping of data items to memory locations. The second stage also determines the number of threads of each process that must be executed.

4.2.1. Process to processor assignment

This stage maps the processes to the processing elements with the objective of maximizing the worst case throughput. This is done assuming an idealistic memory mapping where all non-scalar data items are mapped to the fastest non-local memory, ignoring memory capacities. Each processor is also assumed to run the maximum number of threads.

The ILP formulation incorporates the merge and replication transformations by first assigning processes into "batches", which are then mapped to processors. This is illustrated in Figure 5. In this example, processes A and B are assigned to batch 1, and process C is assigned to batch 2. Assigning multiple processes to a batch is equivalent to merging processes; assigning a batch to multiple processors is equivalent to replicating a process. It can be proved that the batching strategy does not lead to any degradation on the theoretical optimal throughput of an application.

4.2.2. Data to memory mapping

This stage maps the data items to memory locations, and exploits multi-threading to minimize the impact of access latencies. The objective is to achieve a design with maximum throughput.

4.3. Code generation

The designer utilizes the process and data mapping information and generates the functional specification for each core by inlining the application processes, introducing pragmas for specifying data mapping, and the including shared memory communication instructions. The focus of this paper is on system-level design methodology and automated design techniques as specified by the large box in Figure 4.

5. ILP formulations

5.1. Process to processor assignment

The ILP formulation for the first stage merges processes in to batches, and then replicates the batches on to the processors.

Base Variables

Let $\mathcal{B}$ ($|\mathcal{B}| = min(|\mathcal{P}|, |\mathcal{J}|)$ denote the set of batches.

- *Process to batch assignment:* Let $p_{j,b}$, $j \in \mathcal{J}$, $b \in \mathcal{B}$ be $\{0,1\}$ variables; this value will be 1 if process j is assigned to batch b, and 0 otherwise.
- *Batch to processor assignment:* Let $b_{b,p}$, $b \in \mathcal{B}$, $p \in \mathcal{P}$, be $\{0,1\}$ variables; this value will be 1 if batch j is assigned to processor p, and 0 otherwise.
- *Batch replication count:* Let $c_{b,n}$, $b \in \mathcal{B}$, $1 \leq n \leq |\mathcal{P}|$, be $\{0,1\}$ variables; this value will be 1 if batch b is to be replicated on n processors, and 0 otherwise.

Constraints

- *Process to batch assignment:* Every process must be assigned to a single batch. Thus:
$$\forall j \in \mathcal{J} : \sum_{b \in \mathcal{B}} p_{j,b} = 1 \tag{1}$$
- *Processor usage:* Every processor must be assigned a single batch to execute. Therefore:
$$\forall p \in \mathcal{P} : \sum_{b \in \mathcal{B}} b_{b,p} = 1 \tag{2}$$
A batch, however, can be replicated on multiple processors.
- *Batch replication:* Once a batch is selected to be replicated certain number of times say n, then exactly n processors must execute that batch. Thus:
$$\forall b \in \mathcal{B} : \sum_{\forall n} n \cdot c_{b,n} = \sum_{p \in \mathcal{P}} b_{b,p} \tag{3}$$
- *Batch utilization:* A batch must be assigned to one or more processors only if there is at least one process assigned to the batch. Otherwise, the batch can be ignored. As such:
$$\forall b \in \mathcal{B} : \sum_{p \in \mathcal{P}} b_{b,p} \cdot MAX\_VAL \geq \sum_{j \in \mathcal{J}} p_{j,b} \tag{4}$$
where MAX_VAL is a very large value (i.e. $\geq |\mathcal{P}|$).
- *Processor code memory:* The code size of all the processes assigned to a batch cannot exceed the size of the available code memory. Therefore:
$$\forall b \in \mathcal{B} : \sum_{j \in \mathcal{J}} p_{j,b} \cdot S(j) \leq C(p) \tag{5}$$

Objective

Execution of a batch once implies a single execution of all processes assigned to the batch. Thus, the throughput of a process assigned to a batch is equal to the overall throughput of the batch. This is given by the ratio of number of replications of the batch over the summation of the execution times of all processes in the batch. Thus, maximizing the minimum throughput over all batches is equivalent to maximizing the worst case application throughput.

- *Limiting batch:* Let T represent the maximum effective execution time over all batches. Thus:
$$\forall b \in \mathcal{B} : T \geq \sum_{\forall n} \frac{c_{b,n}}{n} \sum_{j \in \mathcal{J}} p_{j,b} \cdot \tau(j) \tag{6}$$

Our objective is to maximize the throughput. Thus:
$$minimize(T) \tag{7}$$

5.2. Multi-threading aware data mapping

The second stage utilizes the results of the previous stage, and generates a mapping of data items to memory locations and a multi-threading configuration.

Constant

- *FIFO and shared memory access:* If a FIFO (or shared memory) is accessed by processes in two different batches, it must be accessible by two different processors. Let ϕ_f, $f \in \mathcal{F}$ (ϕ_s, $s \in \mathcal{S}$) be $\{0,1\}$ variables; this value will be 1 if f (s) is accessed by processes mapped to different batches.

Base Variables

- *Processor thread assignment:* Let $t_{p,k}$, $p \in \mathcal{P}$, $k \in \kappa(p)$ be $\{0,1\}$ variables; this value will be 1 if processor p will run k threads, and 0 otherwise.
- *Data to memory mapping:* Let $a_{d,m}$, $a_{f,m}$, $a_{s,m}$, $d \in \mathcal{D}$, $f \in \mathcal{F}$, $s \in \mathcal{S}$, $m \in \mathcal{M}$ be $\{0,1\}$ variables; this value will be 1 if data item d, FIFO f, or shared memory s is assigned to memory m, and 0 otherwise.

Constraints

- *Thread assignment:* Each processor must be assigned a single multi-threading configuration. As such:
$$\forall p \in \mathcal{P} : \sum_{k \in \kappa(p)} t_{p,k} = 1 \qquad (8)$$
- *Sharing and locality:* The memories local to a processor cannot be accessed by any other processor. Thus, FIFO and shared memories cannot be mapped to these memories if they are accessed by processes assigned to different processors:
$$\forall f \in \mathcal{F}, \forall m \in \mathcal{M} : a_{f,m} + \phi_f + L(m) \leq 2 \qquad (9)$$
$$\forall s \in \mathcal{S}, \forall m \in \mathcal{M} : a_{s,m} + \phi_s + L(m) \leq 2 \qquad (10)$$
- *Memory capacity:* For all memories, the total amount of storage space required to store all the assigned data items cannot exceed the capacity of the memory. Therefore:
$$\forall m \in \mathcal{M} : \sum_{p \in \mathcal{P}} \sum_{k \in \kappa(p)} \sum_{j \in \mathcal{J}} \sum_{d \in \mathcal{D}_l(j)} p_{j,p} \cdot a_{d,m} \cdot t_{p,k} \cdot \sigma(d)$$
$$+ \sum_{f \in \mathcal{F}} a_{f,m} \cdot \sigma(f) + \sum_{s \in \mathcal{S}} a_{s,m} \cdot \sigma(s) \leq C(m) \qquad (11)$$
- *Data assignment:* Every data item must be assigned to a memory location. Therefore:
$$\forall j \in \mathcal{J}, \forall d \in \mathcal{D}_l(j) : \sum_{m \in \mathcal{M}} a_{d,m} = 1 \qquad (12)$$
$$\forall f \in \mathcal{F} : \sum_{m \in \mathcal{M}} a_{f,m} = 1 \quad \forall s \in \mathcal{S} : \sum_{m \in \mathcal{M}} a_{s,m} = 1$$

Objective

We maximize the overall throughput of an application by minimizing the maximum effective execution time of a batch over all processors. This is given by the ratio of the effective execution time of the batch on one processor over the total number of replications of the batch.

- *Effective execution time of a batch on a processor:* Multi-threaded execution leads to two possible situations: i) non-local memory access latency is not completely amortized by multi-threaded execution, or ii) non-local memory access latency is completely hidden by the multi-threaded execution. Let $\tau(b,p)$, be the time between completions of batch $b \in \mathcal{B}$ assigned to processor $p \in \mathcal{P}$.

  When the non-local memory access latency is not completely hidden, we must calculate the total time for a single thread to complete. During this period, each thread will complete once. Thus,
$$\forall b \in \mathcal{B}, \forall p \in \mathcal{P} : \tau_1(b,p) = \sum_{k \in \kappa(p)} \sum_{j \in \mathcal{J}} \sum_{d \in \mathcal{D}(j)} \sum_{m \in \mathcal{M}}$$
$$\frac{t_{p,k}}{k} \left[p_{j,p} \cdot \tau(j) + p_{j,p} \cdot a_{d,m} \cdot \tau_a(m) \cdot N(d) \right] \qquad (13)$$
  In the above equation, we use $\mathcal{D}$ to refer to the set $\mathcal{D}_l(j) \bigcup \mathcal{D}_f(j) \bigcup \mathcal{D}_s(j)$.

Application	Processes	FIFOs	Shared memories	Data Items
Diffserv, IPV4	8	7	1	3
AH Auth., IPv4	8	8	1	5
AH Auth., Diffserv, IPv4	10	10	1	8

Table 2. Application descriptions

  If all memory latency is hidden by multi-threading, we only need to consider the execution time of the processes, and the overhead of accessing local data memory. Thus,
$$\forall b \in \mathcal{B}, \forall p \in \mathcal{P} : \tau_2(b,p) = \sum_{j \in \mathcal{J}} \sum_{d \in \mathcal{D}(j)} \sum_{m \in \mathcal{M}}$$
$$p_{j,p} \cdot \tau(j) + p_{j,p} \cdot a_{d,m} \cdot \tau_a(m) \cdot N(d) \cdot L(m) \qquad (14)$$
  The effective execution time of a batch on a processor is given by the maximum of $\tau_1(b,p)$ and $\tau_2(b,p)$. Thus,
$$\forall b \in \mathcal{B}, \forall p \in \mathcal{P} : \tau(b,p) \geq \tau_1(b,p), \tau(b,p) \geq \tau_2(b,p) \qquad (15)$$
- *Limiting Batch:* The overall throughput of the application is limited by the maximum effective execution time of any batch. Let T represent the effective execution time of the limiting batch:
$$\forall b \in \mathcal{B} : T \geq \sum_{\forall n} \frac{c_{b,n}}{n} \cdot \tau(b,p) \qquad (16)$$

The objective is to maximize the throughput. Thus:
$$minimize(T) \qquad (17)$$

6. Experimental results

We considered three network processing applications that were composed of basic operations that are common in traffic processing of Internet Protocol (IP) networks. We examined combinations of IP Security protocol (IPSec) Authentication Header (AH) verification, Differentiated services (Diffserv) conditioning, and IP version 4 (IPv4) forwarding (as shown in Table 2), mapped to the Intel IXP2400 processor. The applications were profiled using the Intel IXA SDK simulator to obtain the necessary runtime characteristics, using input streams of 64 byte packets.

Architectural constraints of the IXP2400 on the reception and transmission of packets dictated that processes for the respective tasks be distinct and mapped on separate processors. Consequently, two processors were allocated for these operations, and these processes were not considered for further optimizations. Thus, six processors are available to map the remaining application processes. Each FIFO was bounded at 256 bytes, and the abstract shared memory that stores the complete packets was allocated 1MB. The IXP2400 provides a fast interface between the DRAM and media switch fabric, so the abstract shared memory for packets was mapped to the DRAM. Finally, any necessary packet re-ordering is performed by the transmit process.

6.1. Impact of Optimizations

In order to identify the effects of the optimizations we apply, we first generated an optimal solution according to the techniques in [22] to use as a base case. This results in a solution having minimal latency for a single completion of all processes. Data items were then mapped to memories to minimize the latency of the processes. This represents the best possible implementation without taking advantage of the multi-threading or multi-processing capabilities of the hardware architecture. We then applied the process assignment and multi-threading optimizations to this base case, both in isolation and in conjunction. Figure 6 shows the increase

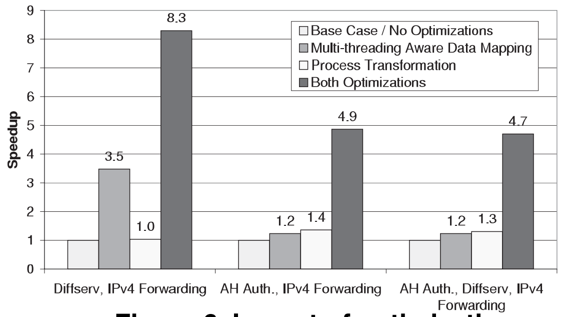


Figure 6. Impact of optimizations

in performance as the optimizations are applied, normalized to the base case as described.

The multi-threading aware data mapping and process assignment optimizations each offer performance improvements, giving on average 2.0× and 1.2× throughput, respectively, when applied in isolation. This improvement is not uniform, however. The Diffserv and IPv4 application is limited in the base case by memory accesses, and thus sees a greater performance increase due to multi-threading aware data mapping. In contrast, the two applications which perform AH authentication are limited by computation time, and thus see greater performance increase due to process assignment optimizations.

Most notable is the substantial performance increase when applying both optimizations. At first glance, the two optimizations appear somewhat independent. It would be expected that their effects would be multiplicative, for an expected average speedup of 2.4× when applying both. The actual average speedup was 6.0×, as the two optimizations aid each other.

Multi-threading aware data mapping minimizes the effect of memory accesses, while process transformations parallelize computation. Merging processes results in a fewer number of processes. As merging two processes combines the memory accesses that each makes, these processes will make more memory accesses. This enhances the optimization possible through multi-threading aware data mapping. Conversely, multi-threading aware data mapping minimizes the effect of memory latency, so that parallelizing the remaining computation has a greater effect. Thus, in order to achieve maximum throughput, process transformation and multi-threading aware data mapping must both be utilized.

6.2. Application Results

We utilized our techniques to map each application to the IXP 2400 architecture, utilizing the full code memory and also constraining the code memory to 400 words. Further, in order to show the accuracy of the throughput estimation, we utilized the IXA simulator to determine the actual throughput of the solutions for the full code memory scenarios. A summary of the run times of the techniques and application throughput is shown in Table 3.

For the most part, each stage was solved very quickly; runtimes were a second or less, with the exception of a few cases of process mapping. Additionally, the results proved to be quite accurate when compared to the simulated implementation. The maximum variance between the expected and actual throughput is just over 10%. In the case of the Diffserv and IPv4 Forwarding application, the bandwidth of the switch fabric was saturated before the full expected bandwidth of the application could be reached.

7. Conclusion

Mapping an application onto a complex multi-processor, multi-threaded network processor is a difficult task. In this paper we presented a two stage ILP formulation for addressing the problem. Experimental results demonstrate that the technique is effectively able to exploit the parallel processing and multi-threading capabilities of the target architecture to achieve high-quality solutions.

Application	Code Memory	Stage 1 Runtime	Stage 2 Runtime	Expected Throughput	Simulation Throughput
Diff., IPv4	Full	126 sec.	$<$ 1 sec.	2440.4 Mbps	*Saturated*
Diff., IPv4	400 inst.	246 sec.	$<$ 1 sec.	2297.9 Mbps	-
AH Auth., IPv4	Full	$<$ 1 sec.	$<$ 1 sec.	188.2 Mbps	167.2 Mbps
AH Auth., IPv4	400 inst.	$<$ 1 sec.	1 sec.	165.2 Mbps	-
AH, Diff., IPv4	Full	$<$ 1 sec.	$<$ 1 sec.	181.8 Mbps	164.8 Mbps
AH, Diff., IPv4	400 inst.	16 sec.	1 sec.	163.2 Mbps	-

Table 3. Application Mapping Results

8. References

[1] L. G. Roberts. Beyond Moore's Law: Internet growth trends. *IEEE Computer*, pages 117–119, 2000.

[2] N. Shah and K. Keutzer. Network Processors: Origin of Species. In *Proceedings of ISCIS XVII, The Seventeenth International Symposium on Computer and Information Sciences*, Oct. 2002.

[3] B. Wheeler, J. Bolaria, and S. Iyer. NPU Market Sees Broad-Based Expansion. *http://www.linleygroup.com/npu/Newsletter/wire050420.html*, Apr. 2005.

[4] E. Johnson and A. Kunze. *IXP2400/2800 Programming: The Complete Microengine Coding Guide*. Intel Press, Hillsboro, OR, USA, 2003.

[5] C. Kulkarni, M. Gries, C. Sauer, and K. Keutzer. Programming Challenges in Network Processor Deployment. In *Int. Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, Oct. 2003.

[6] A. Srinivasan, P. Holman, J. Anderson, S. Baruah, and J. Kaur. Multiprocessor Scheduling in Processor-based Router Platforms: Issues and Ideas. *Workshop on Network Processors at the International Symposium on High Performance Computer Architecture*, 2003.

[7] Z. Tan, C. Lin, H. Yin, and B. Li. Optimization and Benchmark of Cryptographic Algorithms on Network Processors. *IEEE Micro*, 24(5), 2004.

[8] R. Haas et. al. Creating Advanced Functions on Network Processors: Experience and Perspectives. Research Report RZ–3460, IBM, Nov. 2002.

[9] N. Shah, W. Plishker, and K. Keutzer. NP-Click: A Programming Model for the Intel IXP1200. In *Workshop on Network Processors at the International Symposium on High Performance Computer Architecture*, Feb. 2003.

[10] K. Lee, G. Coulson, G. Blair, A. Joolia, and J. Ueyama. Towards a Generic Programming Model for Network Processors. In *Proc. IEEE International Conference on Networks (ICON.04)*, Nov. 2004.

[11] Teja Coproration. "Teja NP Software Platform for the Intel IXP2XXX Network Processor Family". *http://www.teja.com/products/intel_ixp2xxx.html*, June 2006.

[12] Intel Inc. Intel IXA SDK.

[13] R. Ramaswamy and T. Wolf. PacketBench: A tool for workload characterization of network processing. In *Proceedings of IEEE 6th Annual Workshop on Workload Characterization (WWC-6)*, October 2003.

[14] L. Thiele et. al. Design Space Exploration of Network Processor Architectures. In *First Workshop on Network Processors at the 8th International Symposium on High Performance Computer Architecture*, Feb. 2002.

[15] C-T. Hwang, J-H. Lee, and Y-C. Hsu. A Formal Approach to Scheduling Problem in High Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10:464–475, April 1991.

[16] A. Bender. MILP Based Task Mapping for Heterogeneous Multiprocessor Systems. In *Proceedings of the conference on European design automation*, pages 190–197. IEEE Computer Society Press, 1996.

[17] B. Shirazi, K. Kavi, and A. Hurson, editors. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, 1995.

[18] G. Micheli, R. Ernst, and W. Wolf, editors. *Readings in Hardware/Software Co-design*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[19] R. Ramaswamy, N. Weng, and T. Wolf. Application Analysis and Resource Mapping for Heterogeneous Network Processor Architectures. In M. Franklin, P. Crowley, H. Hadimioglu, and P. Onufryk, editors, *Network Processor Design: Issues and Practices*. Morgan Kaufmann, Feb. 2005.

[20] N. Weng and T. Wolf. Profiling and Mapping of Parallel Workloads on Network Processors. In *Proc. of The 20th Annual ACM Symposium on Applied Computing (SAC)*, pages 890–896, Mar. 2005.

[21] V. Ramamurthi, J. McCollum, C. Ostler, and K. Chatha. System Level Methodology for Programming CMP Based Multi-Threaded Network Processor Architectures. In *International Symposium on VLSI*, 2005.

[22] W. Plishker, K. Ravindran, N. Shah, and K. Keutzer. Automated Task Allocation on Single Chip, Hardware Multithreaded, Multiprocessor Systems. In *Workshop on Embedded Parallel Architectures (WEPA-1)*, Feb. 2004.