Report Number: WUCSE-2003-73

2003-11-13

Pipeline Task Scheduling on Network Processors

Mark A. Franklin and Seema Data

Chip Multi-Processors (CMPs) are now available in a variety of systems and provide the opportunity for achieving high computational performance by exploiting application-level parallelism. In the communications environment, network processors (NPs), designed around CMP architectures, are generally usable in a pipelined manner. This leads to the issue of scheduling tasks on processor pipelines. This paper considers problems associated with determining optimal schedules for such pipelines. A system and algorithm called Greedy Pipe is presented. The algorithm employs a greedy heuristic to schedule tasks derived from multiple application flows on pipelines with an arbitrary number of stages. Tasks may be... Read complete abstract on page 2.

Pipeline Task Scheduling on Network Processors

Mark A. Franklin and Seema Data

Complete Abstract:

Chip Multi-Processors (CMPs) are now available in a variety of systems and provide the opportunity for achieving high computational performance by exploiting application-level parallelism. In the communications environment, network processors (NPs), designed around CMP architectures, are generally usable in a pipelined manner. This leads to the issue of scheduling tasks on processor pipelines. This paper considers problems associated with determining optimal schedules for such pipelines. A system and algorithm called Greedy Pipe is presented. The algorithm employs a greedy heuristic to schedule tasks derived from multiple application flows on pipelines with an arbitrary number of stages. Tasks may be shared, and different bandwidths may be associated with each of the application flows. Experimental results indicate that, 95% of the time Greedy Pipe obtains schedules within 10% of optimal. Examples are given to show the use of Greedy Pipe for general pipeline/algorithm design, and for use in the NP environment with typical networking applications.

Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

2003-73

Pipeline Task Scheduling on Network Processors

Authors: Franklin, Mark; Datar, Seema

November 13, 2003

Abstract: Chip Multi-Processors (CMPs) are now available in a variety of systems and provide the opportunity for achieving high computational performance by exploiting application-level parallelism. In the communications environment, network processors (NPs), designed around CMP architectures, are generally usable in a pipelined manner. This leads to the issue of scheduling tasks on processor pipelines. This paper considers problems associated with determining optimal schedules for such pipelines. A system and algorithm called GreedyPipe is presented. The algorithm employs a greedy heuristic to schedule tasks derived from multiple application flows on pipelines with an arbitrary number of stages. Tasks may be shared, and different bandwidths may be associated with each of the application flows. Experimental results indicate that, 95% of the time GreedyPipe obtains schedules within 10% of optimal. Examples are given to show the use of GreedyPipe for general pipeline/algorithm design, and for use in the NP environment with typical networking applications.

Pipeline Task Scheduling on Network Processors [1]

Mark A. Franklin and Seema Datar

Department of Computer Science and Engineering
Washington University in St.Louis, MO, USA
{jbf,seema}@ccrc.wustl.edu

Abstract

Chip Multi-Processors (CMPs) are now available in a variety of systems and provide the opportunity for achieving high computational performance by exploiting application-level parallelism. In the communications environment, network processors (NPs), designed around CMP architectures, are generally usable in a pipelined manner. This leads to the issue of scheduling tasks on processor pipelines. This paper considers problems associated with determining optimal schedules for such pipelines. A system and algorithm called *GreedyPipe* is presented. The algorithm employs a greedy heuristic to schedule tasks derived from multiple application flows on pipelines with an arbitrary number of stages. Tasks may be shared, and different bandwidths may be associated with each of the application flows. Experimental results indicate that, 95% of the time *GreedyPipe* obtains schedules within 10% of optimal. Examples are given to show the use of *GreedyPipe* for general pipeline/algorithm design, and for use in the NP environment with typical networking applications.

1 Introduction

The continuing increase in the logic and memory capacities associated with single VLSI chips has led to the development of Chip MultiProcessors (CMPs) where multiple, relatively simple processors are placed on a single chip. Such CMPs permit the effective exploitation of application level parallelism and thus significantly increase the computational power available to an application. These developments have been exploited by the networking industry where growth in the capacity, speed and functionality of networking infrastructure, have required fast, flexible and efficient computing architectures. This has led to the development of Network Processors (NPs) [1, 2].

Processors within these NPs are often arranged so that they can be used in a pipelined manner. In some cases [2, 3] the NP may contain one or more processor pipelines. A typical architecture that has been analyzed (in a simpler form) from the perspective of obtaining "optimal" power and cache size designs [4, 5, 6] is shown in Figure 1. Packets of information arrive from the network and are classified and routed by a scheduler [7] to one of a number of processor pipeline clusters. The clusters are sized so that bandwidth to off-chip memory meets specified performance requirements, and each cluster contains a number of processor pipelines.

Applications may be associated with one or more of the processor pipelines. The application itself may be pipelined and may utilize multiple processors within a pipeline. A packet, after being routed to a cluster and pipeline, invokes one or more of the applications associated with the pipeline (e.g., routing, compression, etc.) and, after traversing the pipeline, returns to the scheduler where it is transferred to a switch fabric for transmission to the next node in the network.

Given a number of applications that have processor pipeline implementations specified as a series of sequential tasks, one issue that arises is how to assign the tasks to the processors in pipeline so that system throughput is maximized. Applications may also have common tasks and may share one or more pipeline stages. Other design issues relate to determining the number of stages a pipeline must have. While this is clearly related to the line rates one would like to

[1]This research has been supported in part by National Science Foundation grant CCR-0217334.

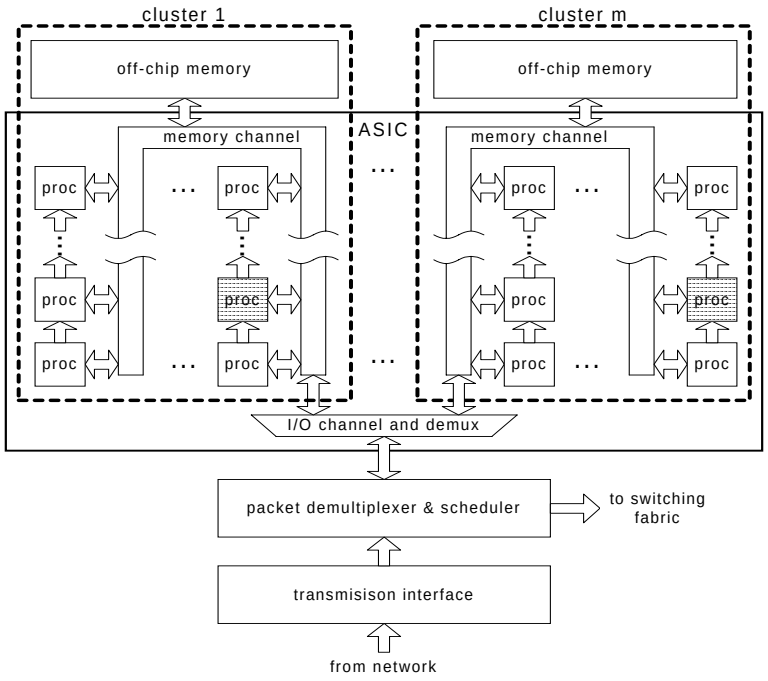

Figure 1: Generic Network Processor

achieve, and the complexity of the applications, there are a number of design options available, each of which requires that a good assignment of application tasks to pipeline stages be obtained. Another example is determining whether algorithmic efforts at changing the number of tasks or task durations associated with an application might help in improving system throughput.

The remainder of this paper presents an approach to solving such assignment problems. We present a mathematical formulation of the problem (Section 2), a greedy algorithm, *GreedyPipe*, that permits rapid and reasonably good assignment solutions (Sections 3 and 4), and the use of *GreedyPipe* in optimizing an NP pipeline design and in aiding in the exploration of alternative application algorithm partitioning (Section 5). Section 6 presents a summary and conclusions.

2 The Pipeline Task Assignment Problem

2.1 Notation and Assignment Constraints

Network processors (NPs) typically have multiple input flows where, in this paper, a flow is defined as a set of successive functions that must be performed on packets belonging to the flow[2].

We consider here application algorithms that may be pipelined and are implemented on a pipeline of identical processors. The general issue of how to develop a pipelined algorithm for a given application is not considered here except for the special cases of Longest Prefix Match (LPM), encryption (AES) and compression (LZW) (discussed in Section 5). Each processor in the pipeline operates on a packet, does some partial processing associated with the application, and then passes the packet (generally modified) along with other information to the next processor in the pipeline. Typically, there are some parts of a flow's processing requirements that are common with other flows and thus they may share the processing that is available on one or more pipeline stages. For example, it is necessary to route all packets and, for a wide class of flows, common routing

[2]Flows correspond to a sequence of functions rather than packets associated with source-destination pairs.

algorithms may be used thus permitting the sharing of a pipeline stage or sequence of stages[3]. After passing through the pipeline, the packet is sent into a switch and from there into the network. We assume that there is a steady infinite stream of arriving packets.

The incoming flows can be represented as the set F where $F = \{F_1, F_2, F_3, \ldots, F_N\}$ and each incoming packet belongs to one of the N flows. The processing associated with each flow can be partitioned into an *ordered* set of tasks, T_{ij}, corresponding to the application(s) requirements of the flow where i ($1 \leq i \leq M_j$), and j ($1 \leq j \leq N$) respectively designate the task and flow number. M_j is the number of tasks associated with flow j. Corresponding to the tasks are times associated with executing the tasks on a given processor stage, t_{ij}. Thus, for flow j:

$$T_j = \{T_{1j}, T_{2j}, T_{3j}, \ldots, T_{M_j j}\} \quad \text{and} \quad t_j = \{t_{1j}, t_{2j}, \ldots, t_{M_j j}\}$$

The pipeline consists of R identical processor stages: $P = \{P_1, P_2, P_3, \ldots, P_R\}$.

The task assignment problem consists of mapping the full set of tasks onto pipeline stages in a manner that preserves task ordering within a flow and optimizes a given performance metric. The real-time assignment problem performs this mapping in real-time based on running estimates of traffic characteristics. In this paper we assume that these characteristics are known apriori and the resulting task assignments are long lived. The assignment of task i from flow j to processor stage k can be expressed using the binary variable X_{ijk} where $X_{ijk} = 1$ if the task is assigned to the processor, and $X_{ijk} = 0$ otherwise. Thus, the number of tasks on a processor k is given by:

$$P_{num.k} = \sum_{j=1}^{N} \sum_{i=1}^{M_j} X_{ijk} \tag{1}$$

Additionally, the following three constraints apply:

- The assignment process must maintain sequential task ordering. Thus, for l, $1 \leq l \leq M_j$, for all i, j, k, r if $X_{ijk} = 1$ and $X_{(i+l)jr} = 1$ then $k \leq r$.
- A task may only be assigned to a single processor. Thus for a given task i from flow j, $\sum_{k=1}^{R} X_{ijk} = 1$.
- Consider situations where the same task is associated with multiple flows. Designating tasks to be shared across flows implies that there will be a single instantiation of the shared task and it will be assigned to a single pipeline stage. Thus, for the case of two flows, j, s, and two tasks, i, r, that are the same and are to share the same stage (and code):

  $$\text{if } T_{ij} = T_{rs} \text{ and } X_{ijk} = 1 \text{ and } X_{rsm} = 1 \text{ then } k = m.$$

  This can be extended naturally to more than two flows. If it is not desired to have such sharing even though the tasks are the same, this can be dealt with by giving the tasks different names.

2.2 Performance Metrics

While the above defines the set of possible legal assignments, to determine an optimal assignment it is necessary to specify a performance metric. The metric of interest in the network processor environment generally relates to maximizing pipeline throughput (i.e., the number of packets per second flowing through the pipeline).

[3]Note that there is generally an overhead associated with moving data between successive stages. This can be easily dealt within the framework provided. However, for a constant overhead between stages, this typically effects the pipeline latency but not throughput and thus does not impact task scheduling.

We consider the case where there are one or more flows that may share tasks and (for simplicity) a single pipeline. Assume that pipeline throughput is limited by the maximum stage execution time taken over all stages in the pipeline. The execution time for a stage is determined by the tasks assigned to each stage. Thus, the execution time for a single flow j, on stage k is given by:

$$s_{jk} = \sum_{i=1}^{M_j} X_{ijk} t_{ij} \tag{2}$$

and the maximum stage execution time for flow j across all the R stages is:

$$P_j = \max_{k=1}^{R}[s_{jk}] \;=\; \max_{k=1}^{R} \left\{ \sum_{i=1}^{M_j} X_{ijk} t_{ij} \right\} \tag{3}$$

The maximum stage execution time over all flows and stages is given by:

$$P = \max_{j=1}^{N} P_j = \max_{j=1}^{N} \;\left\{ \max_{k=1}^{R} \;\left\{ \sum_{i=1}^{M_j} X_{ijk} t_{ij} \right\} \right\} \tag{4}$$

To maximize packet throughput, the problem becomes one of finding a task assignment that minimizes P since the *packet throughput* $\approx 1/P$.

2.3 Related Work

The assignment problem is similar to a number of problems considered in the literature. At the most basic level, a straight forward method of finding the task assignment that minimizes P is to perform a complete enumeration of all possible assignments, identify feasible assignments, and select the optimium one. However, this suffers from a combinatorial explosion of choices. Additionally, optimally efficient solutions are unavailable since the problem has been shown to be NP Hard [8, 9].

There is a long history associated with related problems in deterministic job-shop scheduling [9] and these problems have been investigated from a variety of perspectives including integer programming, use of heuristics and other approaches. Similar problems have also been dealt with in the context of finding compilation techniques for general purpose parallel languages on multiprocessors[10, 11]. The primary objective of the compilation techniques is to minimize the response time while simultaneously reducing overhead due to inter-process synchronization and communication over a general parallel processor.

Real-time packet scheduling problems have also been considered in the context of network processors [4]. In this case, however, packets were assumed to be completely processed on a single processor. A primary concern in that work was to assign packets to processors in a manner that minimized the effect of cold cache misses on performance.

Our work is aimed at maximizing the throughput of a pipelined multiprocessor system by effective assignment of flow tasks to pipeline stages. It differs from prior work in a number of ways. Primarily, the problem definition differs from those considered in the past in that we consider multiple flows and pipelines, sharing of tasks on pipeline stages, and a bandwidth performance metric associated with the requirements of the computer pipeline environment. *GreedyPipe*, the heuristic developed, takes these factors into account in its development.

3 The *GreedyPipe* Algorithm

3.1 Basic Idea

Greedypipe is a heuristic based, in part, on a greedy algorithm and thus gives no guarantee of finding an optimal solution. However, it provides solutions quickly and tests indicate that it finds

near optimal solutions most of the time. Ideally, one would like an assignment where, for each flow, the total execution times of flow tasks associated with each stage are equal. That is, given the total time for executing the tasks associated with flow j is:

$$TotalTime_j = \sum_{i=1}^{M_j} t_{ij} \tag{5}$$

With R stages in the pipeline, as indicated, an optimal allocation of tasks to pipeline stages is one where the execution time for each stage is equal and, under these conditions, the throughput is maximized ($Packet.Throughput_j = 1/Ideal.Delay.per.Stage_j$). Thus, with R stages, for flow j, the ideal delay per stage is:

$$Ideal.Delay.per.Stage_j \;=\; TotalTime_j/R \tag{6}$$

Thus, *Step 1* of the GreedyPipe is to calculate this ideal delay using Equation 6. Actual task times and assignments that satisfy the constraints noted in Section 2.1 will however generally result in unequal execution times associated at each stage. The best of the possible assignments, however, will be the one(s) that come closest to that ideal. Consider the time for execution of all flow j tasks on stage k as given by:

$$t_{jk} = \sum_{i=1}^{M_j} X_{ijk} t_{ij} \tag{7}$$

Since, throughput is calculated from the inverse of the maximum stage execution time, the optimum assignment for flow j is one that minimizes the value of Var_j in the expression given by Equation 8.

$$Var_j = \max_{k=1}^{R} \{[t_{jk}] \;-\; Ideal.Delay.per.Stage_j\} \tag{8}$$

When multiple flows are present there are potentially shared tasks that complicates task assignment. However, various assignments will meet the above constraints and selecting the optimal now requires Equation 8 to be expanded so that the throughput across all the flows is maximized. This can be achieved by selecting the task to stage assignment that minimizes the maximum Var_j across all flows:

$$Var = \max_{j=1}^{N} Var_j = \max_{j=1}^{N} \left\{ \max_{k=1}^{R} \{[t_{jk}] \;-\; Ideal.Delay.per.Stage_j\} \right\} \tag{9}$$

This metric attempts to equalize both the distribution of tasks to stages on a per flow basis and also on an aggregate flow basis. Note that, at a given stage m, potentially there may be multiple allocations for which the minimized Var has the same value. A simple tie breaking algorithm is used that selects the assignment, over all flows, that minimizes the sum of the differences between the ideal delays and the assigned delays.

3.2 Overall Algorithm

The overall heuristic begins by calculating the *Ideal.Delay.per.Stage* for each of the flows. Task to stage allocations start with the first processor stage. Two sets of tasks, satisfying the constraints, are selected from each of the flows for allocation to this processor. The first set is chosen so that the variation, Var_j, given by Equation 8 is minimized and is also a positive number. The second set is chosen similarly, however, Var_j is required to be a negative number. Additional sets that satisfy the constraints may be chosen at the cost of increased complexity and execution time.

At this point, there are two allocations associated with each flow for the first processor and there are thus $(2)^N$ possible combinations of flow allocations. Each of these combinations is examined and the "best" two are kept for use in performing task assignments for the next pipeline stage. The

best two correspond to the two that, for this stage, minimize Var as expressed in Equation 8 with $R = 1$.

Assignments for the next pipeline stage are now considered. The process begins by first calculating new $Ideal.delay.per.stage$ values based on unallocated tasks and the number of remaining pipeline stages. Next, each of the two best allocations from the prior stage is used as a starting point for determining the best task-to-stage assignments for the current stage. For each of these and for each flow, two "best" assignments (positive and negative) are selected. As before, all combinations of these flow assignments are then examined and the two that minimize Var (Equation 9) with $R = 2$ are now kept as starting points for considering the next pipeline stage (Stage 3). This process continues until all stages in the pipeline are examined and a complete assignment has been done. The best of the final stage two assignments is now kept.

Notice that the algorithm has an implicit ordering aspect to it such that tasks and stages are considered in their first-to-last order. While in general this does well, given that local conditions determine allocations at each stage, it will not always result in the optimal allocation. To improve the results one can apply the same heuristic, however, start from the last task and stage, and apply the heuristic in a last-to-first order. Thus, in *GreedyPipe* the algorithm is applied in both directions with the final assignment being the best of the two.

As an example, consider the case of a single three stage pipeline that is to handle two flows having five and four tasks respectively with the flows sharing one task (Task 2). The flows are defined in Table 1. The results of executing *GreedyPipe* are shown in Table 2.

	Task 1	Task 2	Task 3	Task 4	Task 5
Flow 1	T_1	T_2	T_3	T_4	T_5
Task Execution Times	4	2	3	1	3
Flow 2	T_6	T_7	T_8	T_2	
Task Execution Times	5	1	1	2	

Table 1: Two Flows with *ordered* tasks

Stage 1	Stage 2	Stage 3	Max. Stage Time
$T_1, T_6, \quad T = 5$	$T_2, T_3, T_7, T_8 \quad T = 5$	$T_4, T_5 \quad T = 4$	5

Table 2: *GreedyPipe* Task Assignment

3.3 *GreedyPipe* Performance

There are two elements associated with evaluating performance. The first concerns how closely *GreedyPipe* results match the true optimal results. While no analytic bounds on the errors have been developed, extensive experimentation has been performed where the results of *GreedyPipe* were compared with the true optimal as obtained by running an exhaustive search algorithm.

For a wide range of randomly generated conditions, 98% of the time *GreedyPipe* results are within 15% of the optimal and in no case was the result greater than 25% from the optimal. The results however varied with the number of pipeline stages and the percentage of shared tasks associated with different flows. Shared tasks present interesting complications and generally result in somewhat higher errors when the percentage of sharing is greater that 25%.

The second aspect of performance concerns execution time. In the above experiments with systems containing 4 or 5 stages, 1 to 3 flows, and 12 - 15 tasks per flow, exhaustive searches were about three orders of magnitude slower than *GreedyPipe* searchs which took on the order of a second (using a 500Mhz Sparc processor).

4 Pipeline Design with *GreedyPipe*

In systems, such as Network Processors, with multiple pipelines and flows, determining the best pipeline and algorithm partitioning and pipeline stage assignment is difficult. The designer typically has a number of tradeoffs to consider. These include:

- **Number of Pipeline Stages and Number of Pipelines:** Given applications, and associated flows, that have been partitioned into a number of ordered tasks, a designer can select the number of pipeline stages to implement. Up to a point, more stages generally result in higher throughput, however, more stages also requires more chip area and high power consumption. *GreedyPipe* can be used to determine just what throughput can be achieved with a varying number of stages and pipelines.

- **Algorithm Task Sharing:** When multiple flows and associated applications are present, there is often an opportunity for the sharing of applications or of individual tasks across flows. This may result in smaller overall code space being required which, in turn, may reduce the cost of on-chip memory, or increase performance due to reduced cache miss rates. However, when tasks are shared, there is less flexibility in task-to-stage assignments and generally lower overall throughput results. *GreedyPipe* permits fast determination of the performance effects related to task sharing.

- **Algorithm Partitioning:** For many applications, alternative algorithm to task partitionings are possible. For a given pipeline, each partitioning, after assignment, generally leads to different throughput results. *GreedyPipe* can be used to determine those tasks that are performance bottlenecks and what performance gains can accrue from task repartitioning. Up to a point, for a fixed pipeline design, this may result in higher throughput, however, at the cost of algorithm and software redesign.

The sections that follow illustrate the use of *GreedyPipe* in these sorts of design activities. In each subsection, figures illustrating the results of a number of experiments are provided. Each data point presented represents the results of averaging forty experiments. In each experiment *GreedyPipe* was used to generate a task to pipeline assignment and the task times were randomly selected from a uniform distribution ranging from 0 to 10 time units.

4.1 Number of Pipeline Stages:

GreedyPipe may be used to determine the effect of pipeline depth on throughput performance. This is illustrated in Figure 2 where the results for a system with 6 flows, and 20 tasks per flow is shown.

As expected, the throughput increases with the number of stages, and with this many flows and tasks, the increase is almost linear until one reaches fourteen stages. After that, it is more difficult to evenly distribute the tasks over the stages and the throughput asymptotically approaches its maximum of 0.1 (i.e., $1/(maximum\ task\ time)$) This maximum is a result of the fact that it is likely that there is at least one task that is generated that has the maximum time. Similar results are obtained when one plots the throughput as a function of the number of pipelines.

4.2 Sharing of Tasks Between Flows

Task sharing between flows leads to an interdependence between the flows. This may be advantageous and result in better memory utilization and lower instruction cache miss rates, however, it also restricts the number of assignment options and thus potentially reduces the maximum throughput.

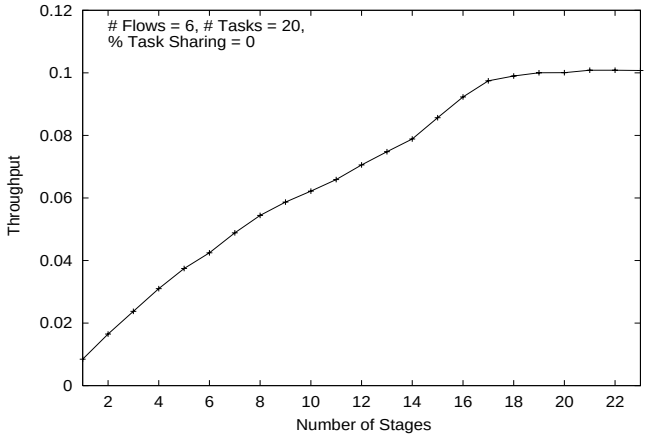

Figure 2: Throughput versus Number of Pipeline Stages

Experiments were conducted for the case of 6 flows, 20 tasks per flow and a single 8 stage pipeline where the fraction of tasks for each flow that are shared with other flows was varied. Thus, a 50% level of sharing means that 50% of tasks for each of the flows are common with the other flows. As expected, the results (Figure 3) indicate a significant decrease in throughput as more tasks are shared between flows. The decrease is over 35% when one moves from 0% sharing to 100% sharing. In a full design analysis, this would be balanced against the potential gains noted above.

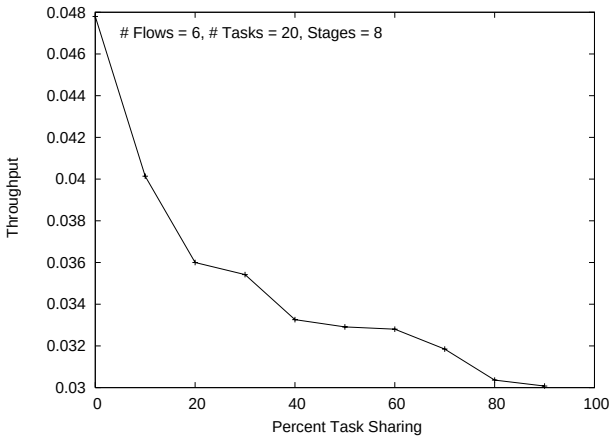

Figure 3: Throughput vs % Shared Tasks

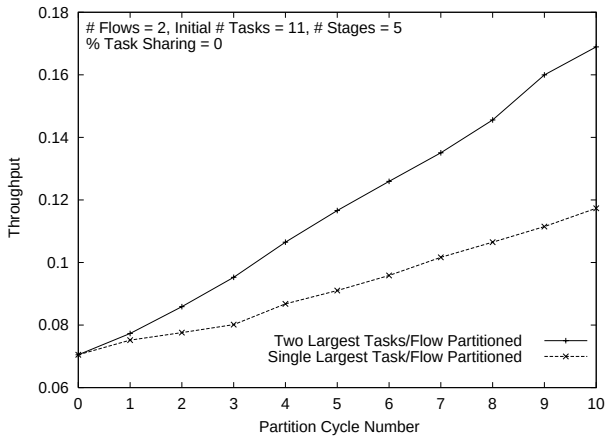

Figure 4: Throughput vs Task Partitioning

4.3 Task Partitioning

For many applications that can be implemented in a pipelined manner, there is a choice concerning the task partitioning of the application. Having more tasks generally results in both having greater flexibility in assigning the tasks to the hardware pipeline and in being able to use longer pipelines. This usually results in higher throughput.

However, there are two potential drawbacks. First, it can be difficult to divide tasks beyond some basic application partitioning and thus there may be a nontrivial personnel cost associated with this job. Second, greater task partitioning often results in larger inter-task communications

costs that may increase latency and reduce throughput. However, in order to make a judgement as to whether increased task partitioning is worthwhile considering, it is first necessary to determine the potential performance gains that might result from such an endeavor. *GreedyPipe* permits one to consider the possible gains from additional task partitioning.

The effects of task partitioning on throughput are illustrated in Figure 4. For both curves presented, the experiments had 2 flows, 5 tasks per flow, a single 5 stage pipeline, and no task sharing. With the lower curve, the longest task in each flow is successively divided into two equal tasks and then *GreedyPipe* is used to find a new task assignment. The "Partition Cycle Number" corresponds to how many times this division has occurred (a new maximum task is determined and divided on each cycle). With the upper curve, the two longest tasks in each flow are divided in a similar manner and the throughput is obtained. Both cases are beneficial since both provide more opportunities for improved task assignments that aim at equalizing the delay associated with each stage (and thus maximize throughput). This permits the designer to determine the potential benefit associated with spending more time on algorithm partitioning.

5 A Network Processor Problem

The Intel IXP 2800 is an example of an NP that can potentially be configured as a set of processor pipelines where the total number of processor stages is sixteen. Consider now a workload where there are the following three flows:

1: Longest Prefix Match (LPM) - A flow where the only function to be performed is that of routing the packet using the LPM algorithm.

2: LPM & Compression - A flow where the NP must perform LPM and also compression on the packet payload.

3: LPM & Encryption - A flow where the NP must perform LPM and also encryption on the packet payload.

Software implementations for LPM, Compression and Encryption are available and may provide adequate performance at the edges of the network. However, as one moves towards the core of the network, real-time constraints generally requires the use of fast special purpose hardware. An alternative to providing such special hardware is to utilize a general processor pipeline for these applications. As discussed later, pipelined implementations of these functions are available.

Say that our objective is to maximize overall throughput given the number of available pipeline processor stages. [4] Given general pipelined implementations of the functions, and their associated task times, the design space for maximizing throughput includes the following choices:

1. Number and length of pipelines: Both the number of pipelines and the length of each pipeline can be selected subject to the constraint that the total number of processors over all the pipelines must be $\leq$ the number of processors available.

2. Number of tasks per function: Given a general approach to implementing each of the functions as a software pipeline of ordered tasks, just how should a particular set of tasks be selected.

[4] Two important issues are omitted in this discussion. First, we are not considering multithreading effects. However, if we assume a high enough level of multithreading so that all off-chip memory latencies are masked, then our rough throughput analysis is not significantly effected by this assumption. Second, we are not considering other issues associated with the memory hierarchy. That is, we are assuming that contention for common resources is not appreciable. Extensions to this work will bring these effects into the model.

3. Assignment of function tasks to pipeline stages: Given the two items above, assign each of the tasks to the pipeline stages in a manner that maximizes the overall throughput of the NP.

With *GreedyPipe* it is a relatively simple matter to explore key aspects of this design space. Given a pipelined function implementation (item 2) and a choice of number and length of pipelines (item 1), *GreedyPipe* will choose a near optimal assignment of tasks to stages (item 3). One can then iterate over set of allowable pipeline configurations (item 1) and obtain a near optimal overall design. We next review the flow functions and some pipelined implementation options.

5.1 Longest Prefix Matching (LPM)

Performing IP address lookup for packet routing is a key operation that must be performed by routers and such lookups require that a Longest Prefix Matching (LPM) algorithm be executed [13]. Because of its central role[12], NPs often include facilities to perform fast IP prefix matching often using a combination of software and special purpose hardware. One may also implement LPM utilizing a processor pipeline. Such an approach potentially has high and additionally may also be modified to meet the requirements of evolving standards. This section considers a pipelined LPM algorithm based on the work of Moestedt and Sjodin [13]. In their paper, a dedicated pipeline of special purpose hardware is presented. Our implementation uses a pipeline of general purpose processors and is based on the development of a routing tree that contains three types of nodes:

- **Valid Route Nodes**: Tree leaf nodes that correspond to legal or valid routes (or destinations). Associated with these nodes are the router output port information.
- **Invalid Route Nodes**: Tree leaf nodes that correspond to invalid routes.
- **Part Route Nodes**: Tree interior that represent branching nodes in the tree and correspond to part of a prefix.

Consider an example (Figure 5) containing three prefixes embedded within a three level tree. The first level, leaf nodes labelled (1), is of length 3 and corresponds to the prefix 001. The second, leaf nodes labelled (2), is of length 7 and corresponds to prefix 0010001. The third, leaf nodes labelled (3) is of length 3 and corresponds to prefix 110. Note that with this LPM algorithm smaller prefixes (e.g., 001) that are themselves contained in longer prefixes (e.g., 0010001) may spawn additional levels and Valid Route Nodes (e.g., level 3 nodes corresponding to the first prefix). Constructing the tree itself requires that one first decide on the number of levels desirable, and the number of bits to be considered at each level. Thus, there are numerous tree and associated data structures that satisfy the routing table requirements and have differing memory requirements.

Given a packet destination address and the above routing tree structure, obtaining the route involves successively accessing the tree following the path associated with this address. Say we have the address [0010 0111 X] where X corresponds to the remaining 24 bits in a 32-bit address. Reaching leaf node (2) in Level 3 requires three tree lookups following the wide path in Figure 5.

An approach to pipelining such an algorithm is to associate each level lookup with a separate task and allocate these tasks to stages in a processor pipeline. In the above example a three processor pipeline would be used, one stage for each level. With sufficient data memory bandwidth, this would increase the throughput of a packet stream by a factor of three over a single processor implementation. Alternatively, two of the levels could be combined and implemented on a pipeline stage resulting in a two stage pipeline. This might be desirable if many of the lookups terminated at a given stage (as is often the case) thus resulting in fewer lookups in later stages. Using the earlier terminology, this example contains a single flow consisting of three computationally identical

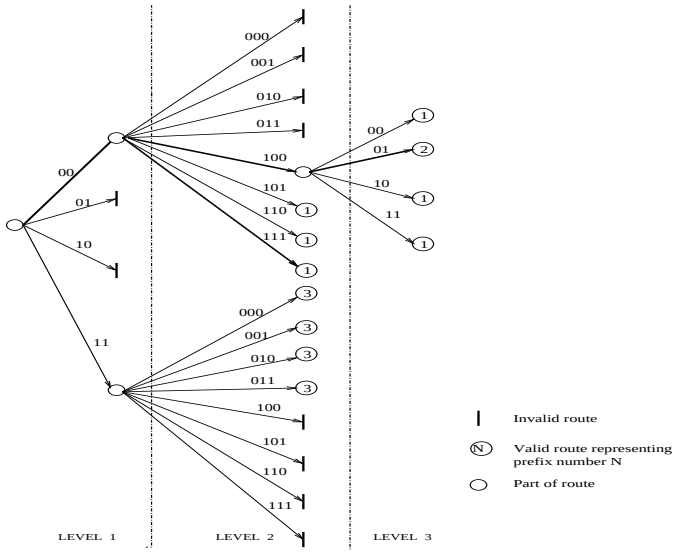

Figure 5: Example Longest Prefix Match Tree

tasks where differences in execution times result from the number of memory accesses associated with each of the processor stages.

	# Tasks	Task 1	Task 2	Task 3	Task 4	Task 5
LPM	5	2.8×10^{-2}	4.0×10^{-2}	2.6×10^{-2}	2.0×10^{-2}	1.4×10^{-2}
ENCRYPTION	11	17.4	$\approx$ 11.4 per task for up to 10 tasks			
COMPRESSION	15	$\approx$ 9.44 per task for up to 15 tasks				

Table 3: Task Times (μsec) for LPM, Encryption & Compression

A tool called *SimplePipe* [5] was used to evaluate tree level/task to pipeline stage assignments for a problem having 117,212 prefixes obtained from a Sprint network router (AS1239) [17, 18]. From this set, a five level tree was constructed with each successive level handling 14, 5, 3, 2 and 8 bits. The entire tree contains 504,857 nodes. Traffic was modelled as a set of 5,000 successive routing requests where the distribution of request prefixes followed those empirically obtained in [19, 20].

With a five level tree, a separate task (with its task time) may be associated with accessing each tree level. These times are given in Table 3. Initially, *GreedyPipe* was used to obtain the best assignment of these five tasks to processor pipelines of different lengths (1 to 16 stages). This is shown in the top graph in Figure 6 where, at four stages, the maximum throughput is achieved (note that task 2 takes the longest time) and remains constant after that.

5.2 AES Encryption - A Pipelined Implementation

Encryption involves transforming unsecured information ("plaintext") into coded information ("ciphertext") with the process being controlled by an algorithm and a key. The Advanced Encryption

[5] *SimplePipe* is a pipeline simulation tool based on SimpleScalar [16, 14]. An in-order processor with a clock rate of 1 Ghz was modelled. All processor stage caches were taken to be of equal size with the instruction cache being sufficiently large to hold the entire stage program. A 4-way associative, 8KB data cache was assumed (No L2 cache was present) with off-chip memory latencies set at 20 processor clock cycles. The off-chip memory was taken to be structured as a set of independent banks, one for each of the processor stages where each bank holds data associated with the tasks assigned to it. A fixed stage-to-stage communications delay of 10ns was also assumed. Data for encryption and compression was obtained directly from SimpleScalar with the same parameters indicated above.

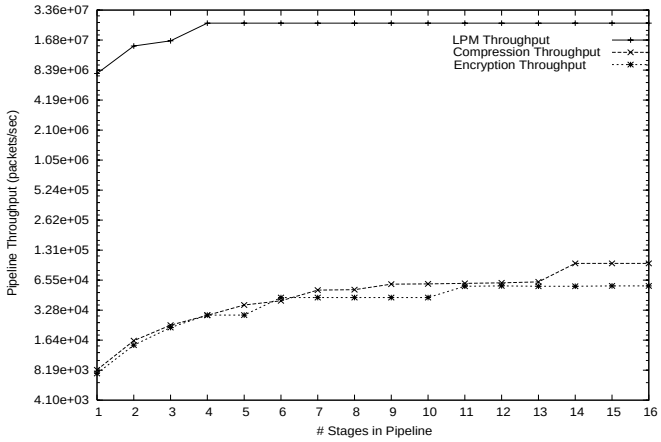

Figure 6: Throughput vs. Number of Stages (A single separate pipeline for each application)

Standard (AES) is considered here along with the Rijndael encryption algorithm. Rijndael is an iterated block cipher which supports independently specified variable block and key lengths (128, 192 or 256 bits). The algorithm has a variable number of iterations dependent on the key length. For 128 bit blocks, considered here, 10 iterations are required [21].

Algorithm transformations operate on an intermediate result called the State which is defined as a rectangular array with 4 rows and N_b columns ($N_b = block.length/32$). Transformations treat the 128-bit data block as a 4 column rectangular array of 4-byte vectors. The data block along with the 128-bit (16B) plaintext is interpreted as a State. The cipher key is also considered to be a rectangular array with four rows, the number of columns N_k being the key length divided by 32. The number of rounds (iterations) depends on the values N_b and N_k and, for 128 bit blocks, is 10.

The algorithm consists of an initial data/key addition, then 9 round transformations followed by a final round. The Key schedule expands the key entering for the cipher so that a different round key is created for each Round Transformation. Such transformations are comprised of the following four operations: *ByteSub* Transformation, a *shiftRow* Transformation, a *MixColumn* Transformation and a *Round Key Addition*. The final round is similar to earlier rounds except that the *MixColumn*(State) operation is omitted. Details associated with these operations can be found in [22]. An overview of a single processor/stage implementation is shown in on the left side of Figure 7 while a pipelined implementation employing loop unrolling [15] is shown on the right side of the figure.

As indicated, each iteration or can be associated with a pipelined task. The task times associated with each of the eleven tasks are shown in Table 1. The values were obtained using from a SimpleScalar simulation of the algorithm. Figure 6 shows the throughput obtained when *GreedyPipe* is used to assign only these eleven tasks to pipelines of different lengths. Notice that since there are eleven tasks, after the pipeline length reaches a length of eleven there is no throughput improvement. Furthermore, note that given the much higher computational complexity of encryption versus LPM, the throughput that can be achieved is between two and three orders of magnitude less for pipelines of equivalent length.

5.3 Data Compression - A Pipelined Implementation

With data compression (DC) a string of characters is transformed into a new reduced length string having the same information. In this way the bandwidth required for passing a given string is reduced. *LZW* is a DC method that takes advantage of recurring patterns of strings that occur in

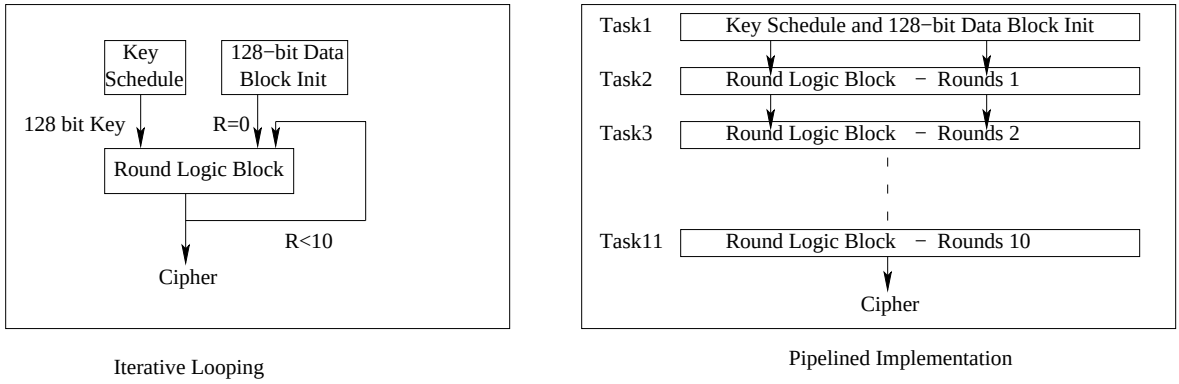

Figure 7: Rijndael Algorithm Implementation Block Diagram

data files. The original LZW method was created by Lempel and Ziv [23] and was further refined by Welch [24].

LZW is a "dictionary"-based compression algorithm that encodes input data by referencing a dictionary. Encoding a substring only requires that a single code number corresponding to that substring's dictionary index be written to the output file. *LZW* starts out with a dictionary of 256 characters (for the 8-bit case) and uses those as the "standard" character set. Data is then read 1 byte at a time (e.g., 'p', 'q', etc.) and encoded as the index number taken from the dictionary. When a new longer substring is encountered (say, "pq" and later saay "pqr"), the dictionary is expanded to include the new substring and an index is associated with the new substring. The new index is then used when the new substring is encountered. To quickly associate substrings with indcies, hashing techniques may be employed and to to limit memory size, a limit placed on maximum number of dictionary entries (say, 1024).

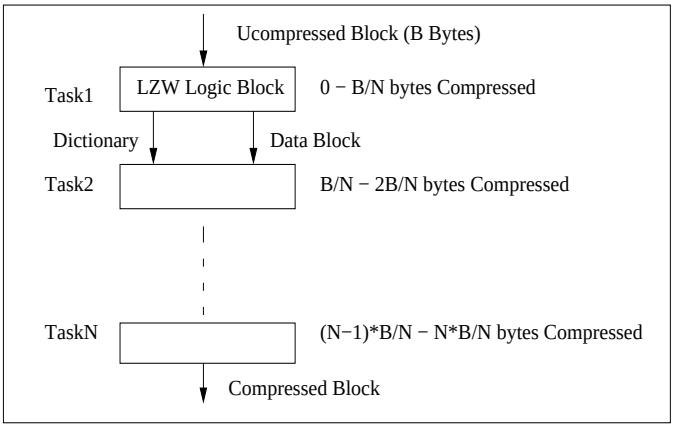

Figure 8: Pipelined Implementation of the *LZW* Algorithm

With a straight forward pipelined LZW implementation the input data is partitioned and successive pipeline stages operate on successive portions of the input data. Thus, if there are N stages in the pipeline, and a block of B bytes requires compression, each stage operates and compresses B/N successive input data bytes. Figure 8 shows the pipelined implementation. The partially compressed data block and the updated dictionary is made available to every stage of the pipeline and the resulting system throughput increases almost linearly with N.

Task times (found using Simple Scalar) associated with each task for a fourteen task implemen-

tation are shown in Table 1 and Figure 6 shows the throughput obtained when *GreedyPipe* is used to assign these tasks to pipelines of different lengths.

5.4 *GreedyPipe* NP Example Design Results

The results shown in Figure 6 are for single flows executing on a single pipeline. The problem becomes more complex when:

1. Multiple flows are considered with some flows requiring multiple applications.
2. Some of the flow applications/tasks are shared and are to be instantiated only once.
3. Multiple processor pipelines are present with a constraint on the total number of stages available.

This more complex problem is now considered. Table 3, shows the execution times per task for the three applications where the ouput from one task becomes the input to the next task. Assume that we have the three flows indicated earlier. We assume that packets associated with Flow 1 (LPM) are 40 Bytes long while packets from Flows 2 and 3 (LPM & Encryption; LPM & Compression) are 1,500 Bytes long. Four experiments were performed:

1. **Single Pipeline:** A single 16 stage pipeline was considered where the LPM application was shared among the three flows.
2. **Two Pipelines:** Two pipelines were used where the length of each pipeline was varied between 1 and 15 stages, with the sum of the stages being set 16. Flow 1 was assigned to one pipeline and its LPM application was not shared with the other Flows. Flows 2 & 3 were assigned to the second pipeline and the LPM application was shared between them.
3. **Three Pipelines:** Three pipelines were used, one flow assigned to each, where the length of each pipeline was varied between 1 and 14 stages with the sum of the stages set to 16.

Consider the first case above where there is a single 16 stage pipeline. Entering the set of flows, their respective applications, and the application task sequence and times, *GreedPipe* performs the assignment process and obtains an assignment that maximizes the total throughput. The results in this case are that stage 1 is assigned to LPM (shared across all flows) while the remaining stages are shared by the encryption and compression components of flows 2 and 3. Table 4 shows the resulting best overall bandwidth (both in Gbps and Packets/second, Pps) for each flow, and the length of each pipeline for the two and three pipeline cases. Note that with a single pipeline, the bandwidth is constrained by the longest latency task which, in this case, corresponds to encryption which has the highest computational complexity.

Number Pipelines	Flow 1 Rate (Gbps / Pps)	Flow 2 Rate (Gbps / Pps)	Flow 3 Rate (Gbps / Pps)	Pipe 1 Length	Pipe 2 Length	Pipe 3 Length
1	0.018 / 5.7×10^4	0.68 / 5.7×10^4	0.68 / 5.7×10^4	16	0	0
2	7.94 / 2.4×10^7	0.68 / 5.7×10^4	0.68 / 5.7×10^4	4	12	0
3	7.94 / 2.4×10^7	0.525 / 4.3×10^4	0.48 / 4.0×10^4	4	6	6

Table 4: Bandwidths for Best Assignments (Pps=Packets/second; Flow 1 packet length=40 Bytes; Flows 2 & 3 packet length=1500 Bytes)

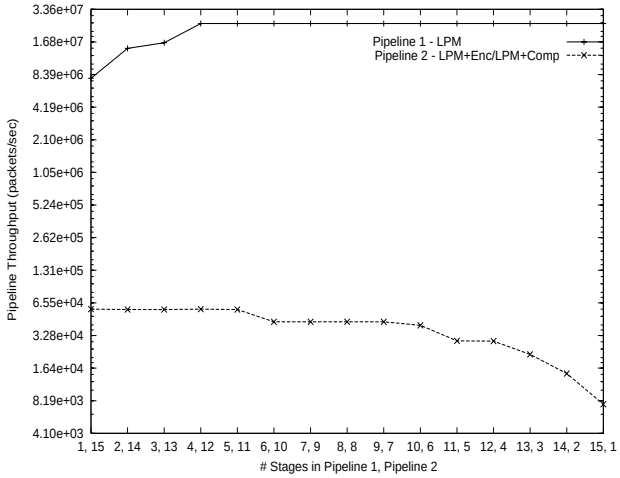

Figure 9: Two Pipelines - Throughput vs. Num. Stages (X, Y –> X stages for Pipe 1 & Y stages for Pipe 2)

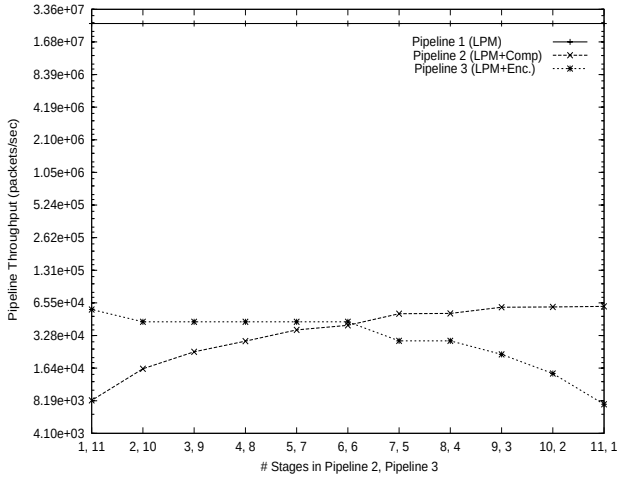

Figure 10: Three Pipelines - Throughput vs Num. stages for Pipelines 2 & 3; Pipeline 1=4 stages

In the second case the 16 stages are divided into two pipelines with their lengths not necessarily being equal. *GreedyPipe* was executed iteratively with a different number of stages assigned to each pipeline and, for each pipeline length selection, a near optimal assignment obtained (see Figure 9). For pipeline 1 (Flow 1, LPM), the results are the same as seen in Figure 6 since there is a single pipeline associated with that flow. As the number of stages for pipeline 1 increases however, the number remaining for pipeline 2 decreases. The result is that there are increased delays for flows 2 (LPM & Encryption) and 3 (LPM & LZW). *GreedyPipe* yields the near optimum task allocations for each of these pipeline lengths. Note that for pipeline 2, the bandwidth is dominated by the requirements of encryption. Flow 1 bandwidth is significantly improved since, having its own pipeline, its packets are now not limited by the delays for encryption.

For the third case, the 16 processor stages are shared across three pipelines with flows 1, 2 and 3 being assigned to pipelines 1, 2 and 3 respectively. Using *GreedyPipe*, all partitioning of the 16 stages across 3 pipelines were considered and evaluated. The results indicate that assigning 4 stages to Pipeline 1 (Flow 1, LPM) achieves the highest througput. Figure 10 plots results associated with pipeline 1's length equal to 4 stages. As more of the remaining 12 stages stages are assigned to flow 2 there are fewer for flow 3. Given the pipelined implementations of encryption and compression, the crossover point on the graph corresponds to the highest throughput result (also see Table 4). Due to the fact that the length of pipelines 2 and 3 are constrained, the full effect of pipelining the flows 2 and 3 cannot be realized and the maximum throughput for those flows is less than the two pipeline case.

References

[1] IBM Corp., "IBM Power Network Processors." `http://www-3.ibm.com/chips/products/wired/products/network_processors.html`, 2001.

[2] Intel Corp., "Intel IXP 2800 Network Processor." http://developer.intel.com/design/network/products/npfamily/ixp2800.htm., 2003.

[3] J. Marshall, "Cisco Systems - Toaster2," in *Network Processor Design, Vol.1, by P. Crowley, M. Franklin, H. Hadimioglu, and P. Onufryk*, San Francisco, CA.: Morgan Kaufmann Publishers, Inc., 2003.

[4] T. Wolf and M. A. Franklin, "Locality-Aware Predictive Scheduling of Network Processors," in *Proc. 2001 IEEE Inter. Symp. on Performance Analysis of Systems & Software*, (Tucson, Arizona), Nov. 2001.

[5] M. Franklin and T. Wolf., "A network processor performance and design model with benchmark parameterization," in *Network Processor Design, Vol.1 (Also in Network Processor Workshop (HPCA-8), Cambridge, MA., Feb 2002)*, San Francisco, CA.: Morgan Kaufmann Publishers, Inc., 2003.

[6] M. Franklin and T. Wolf., "Power considerations in network processor design," in *Network Processor Design, Vol.2 (Also in Network Processor Workshop (HPCA-9), Anaheim, CA., Feb 2003)*, San Francisco, CA.: Morgan Kaufmann Publishers, Inc., 2003.

[7] T.Wolf and M. Franklin, "Design tradeoffs for embedded network processors," in *Proc. of Inter. Conf. on Architecture of Computing Systems (ARCS) (Lecture Notes in Computer Science)*, vol. 2299, (Karlsruhe, Germany), pp. 149–164, Apr 2002.

[8] P. Chretienne, J. E. G. Coffman, J. K.Lenstra, and Z. Liu, *Scheduling Theory and its Applications*. Chichester, England: John Wiley & Sons, 1995.

[9] A. S. Jain and S. Meeran, "Deterministic Job-Shop Scheduling: Past, Present and Future," *European Jrnl. of Operational Research*, vol. 113, no. 2, 1999.

[10] V. Sarkar and J. Hennessey, "Compile-time Partitioning and Scheduling of Parallel Programs," in *In ACM SIGPLAN '86 Symp. on Compiler Construction*, pp. 17–26, 1986.

[11] M. Schwehm and T. Walter, "Mapping and Scheduling by Genetic Algorithms," in *Conf. on Algorithms and Hardware for Parallel Processing*, pp. 832–841, 1994.

[12] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high-speed prefix matching," *ACM Transactions on Computer Systems*, vol. 19, Nov. 2001.

[13] A. Moestedt and P. Sjodin, "IP Address Lookup in Hardware for High-Speed Routing," in *Hot Interconnects*, August 1998.

[14] M. Franklin and V. Joshi, "SimplePipe: A Simulation Tool for Task Allocation and Design of Processor Pipelines with Application to Network PRocessors," tech. rep., Washington Univ. in St. Louis, CSE Dept, (Pending).

[15] P. Chodowiec and P. Khuon and K. Gaj, "Fast implementations of secret-key block ciphers using mixed inner- and outer-round pipelining," in *ACM SIGDA Inter. Symp. on Field Programmable Arrays (FPGA'01)*, (Monterey, CA), Feb. 2001.

[16] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modelling," in *IEEE Computer*, February 2002.

[17] "BGP Table Data." http://bgp.potaroo.net/, 2003.

[18] "AS1239 BGP Table Data." http://bgp.potaroo.net/1239/bgp-active.html, 2003.

[19] K. Claffy, G. Miller, and K. Thompson, "The Nature of the Beast: Recent Traffic Measurements from an Internet Backbone," tech. rep., April 1998.

[20] "CAIDA : The Cooperative Association for Internet Data Analyses ." `http://www.caida.org`.

[21] "AES Algorithm Rijndael Information." http://csrc.nist.gov/CryptoToolkit/aes/rijndael.

[22] V. Rijmen and J. Daemen, "The Block Cipher Rijndael," in *Proc. of the third international conference on smart card research and applications, CARDIS'98, LNCS 1820*, pp. 277–284, 2000.

[23] J. Ziv and A. Lempel, "Compression of Individual Sequences via Variable-Rate Coding," *IEEE Trans. Information Theory*, vol. 24, no. 5, pp. 530–536, 1978.

[24] T. A. Welsh, "A Technique for High-Performance Data Compression," *Computer*, vol. 17, no. 6, pp. 8–19, 1984.