Report Number: WUCSE-2003-60

2003-06-27

# Task Scheduling of Processor Pipelines with Application to Network Processors

Seema Data and Mark A. Franklin

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) are often designed around CMP architectures and in this context the processors may be used in a pipelined manner. This leads to the issue of scheduling tasks on processor pipelines. This paper considers problems associated with determining optimal application task assignments for such pipelines. A system and algorithm called Greedy Pipe is presented and its performance analyzed. The algorithm employs a greedy heuristic to schedule tasks derived from... **Read complete abstract on page 2.**

# Task Scheduling of Processor Pipelines with Application to Network Processors

Seema Data and Mark A. Franklin

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) are often designed around CMP architectures and in this context the processors may be used in a pipelined manner. This leads to the issue of scheduling tasks on processor pipelines. This paper considers problems associated with determining optimal application task assignments for such pipelines. A system and algorithm called Greedy Pipe is presented and its performance analyzed. The algorithm employs a greedy heuristic to schedule tasks derived from multiple application flows on pipelines with an arbitrary number of stages. Tasks associated with multiple applications may also be shared. Experimental results indicate that over a wide range of conditions, 95% of the time Greedy Pipe quickly obtains schedules within 10% of optimal.

Department of Computer Science & Engineering

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

2003-60

# Task Scheduling of Processor Pipelines with Application to Network Processors

Authors: Franklin, Mark ; Datar, Seema

June 27, 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) are often designed around CMP architectures and in this context the processors may be used in a pipelined manner. This leads to the issue of scheduling tasks on processor pipelines. This paper considers problems associated with determining optimal application task assignments for such pipelines. A system and algorithm called GreedyPipe is presented and its performance analyzed. The algorithm employs a greedy heuristic to
schedule tasks derived from multiple application flows on pipelines with an arbitrary number of stages. Tasks associated with multiple applications may also be shared. Experimental results indicate that over a wide range of conditions, 95% of the time GreedyPipe quickly obtains schedules within 10% of optimal.

# Task Scheduling of Processor Pipelines with Application to Network Processors [1]

Seema Datar and Mark A. Franklin

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) are often designed around CMP architectures and in this context the processors may be used in a pipelined manner. This leads to the issue of scheduling tasks on processor pipelines. This paper considers problems associated with determining optimal application task assignments for such pipelines. A system and algorithm called *GreedyPipe* is presented and its performance analyzed. The algorithm employs a greedy heuristic to schedule tasks derived from multiple application flows on pipelines with an arbitrary number of stages. Tasks associated with multiple applications may also be shared. Experimental results indicate that over a wide range of conditions, 95% of the time *GreedyPipe* quickly obtains schedules within 10% of optimal.

# 1 Introduction

Two general trends have led to the need for effective scheduling of processor pipelines. The first relates to the continuing increase in the logic and memory capacities associated with single VLSI chips. This has led to the development of Chip MultiProcessors (CMPs) where multiple (up to several dozen) relatively simple processors have been placed on the same chip. Use of such CMPs permit the effective exploitation of application level parallelism and thus potentially significantly increase the computational power available to an application.

The second general trend relates to growth in the networking infrastructure, the requirements for new networking functions, and increases in line bandwidths particularly within the network core. This has resulted in a need for fast, flexible computing architectures

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

dedicated to efficiently performing networking functions, yet flexible enough to respond to changes in protocol standards and functional requirements. These requirements have led to the development of Network Processors (NPs) [1, 2, 3] that are often combinations of CMPs augmented with special purpose logic.

Processors within these NPs are often arranged so that they can be used in a pipelined manner. In some cases [3] the NP may contain several 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] is shown in Figure 1. Packets of information arrive from the network and are classified and routed by a scheduler [6] to one of a number of processor pipeline clusters. The clusters are sized so that bandwidth to off-chip memory is adequate for reasonable performance and each cluster contains a number of processor pipelines.

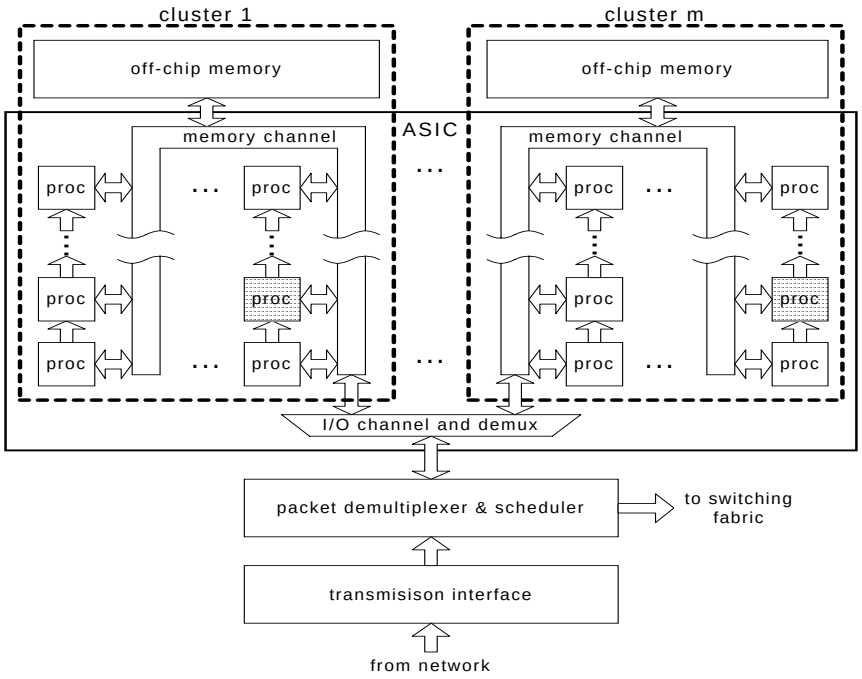


Figure 1: Generic Network Processor

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, will invoke one or more of the applications asso-

ciated with the pipeline (e.g., routing, compression, encryption, etc.) and, after traversing the pipeline, returns to the scheduler where it will typically be sent 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 just how to assign the tasks to the processors in a pipeline such that system throughput is maximized. Applications may also have common tasks and may share the same pipeline stage. 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 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 achieved. Another example is determining whether algorithmic efforts at changing the number of tasks associated with an application or their individual durations would help in improving system throughput.

The remainder of this paper presents an approach to solving the task assignment problem. We describe and analyze a novel greedy algorithm call *GreedyPipe* that permits rapid and reasonably good solutions to the task assignment problem. Additionally, the paper shows how *GreedyPipe* can be used in optimizing pipeline design and in aiding the exploration of alternative application algorithm partitionings.

Section 2 that follows casts the problem in a formal mathematical context. Section 3 presents some of the related work in this area, considers task scheduling/assignment strategies and introduces *GreedyPipe*. The section also presents a simple example and discusses *GreedyPipe* performance. Section 4 illustrates how *GreedyPipe* can be used in exploring the design space to determine, for example, the optimum number of pipeline stages. Section 5 presents a summary and conclusions.

# 2 The Pipeline Task Assignment Problem

## 2.1 Notation and Assignment Constraints

Network processors typically have multiple input flows where each flow represents a connection between designated source and destination in the network. Flows consist of a sequence of packets that are separated in time. Each flow has certain characteristics and processing demands. These consist of requests that certain application algorithms be applied to the successive packets associated with the flow.

We consider here application algorithms that may be pipelined and are implemented on a pipeline of identical processors. The difficult issue of how to develop a pipelined algorithm for a given application is not considered here. However, this has been studied for many networking and related media applications [7, 8, 9]. 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 segments. For example, it is necessary to route all packets and, for a wide class of flows, common routing algorithms may be used, thus permitting the sharing of a pipeline stage or sequence of stages[2]. 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 stream of packets arriving (i.e., an infinite supply of work to be done).

The incoming flows can be represented as the set $F$ where

$$\mathrm{F} = \{F_1, F_2, F_3, \ldots, F_N\}$$

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

[2]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.

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. Thus, for flow $j$:

$$T_j = (\ T_{1j},\ T_{2j},\ T_{3j},\ \ldots\ ,T_{M_j j}\ )$$

Each task belonging to a flow has a processor execution time given by:

$$t_j = \{t_{1j},\ t_{2j},\ \ldots\ ,t_{M_j j}\} \text{ where } j = 1 \text{ to } N.$$

The pipeline itself consists of $R$ identical processor stages:

$$P = \{P_1,\ P_2,\ P_3,\ \ldots\ , P_R\}$$

The task allocation or assignment problem consists of mapping the full set of tasks onto the stages of a pipeline in a manner that preserves task ordering within a flow and optimizes a given performance metric.

Assignment of task $i$ from flow $j$ to processor $k$ 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 $l,\ 1 \leq l \leq M_j$,

$$\text{for all } i, j, k, r \quad \text{if } \ X_{ijk} = 1 \text{ and } X_{(i+l)jr} = 1 \quad \text{ then } \ k \leq r.$$

- A task may only be assigned to a single processor. Thus for a given task $i$ from flow $j$, $\sum_{k=1}^{R} X_{ijk} = 1$.

- For this analysis, an additional constraint is applied in 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 and two tasks:

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

  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 easily by giving the tasks different names.

## 2.2 Performance Metrics

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

We consider the case where there are one or more flows, flows that may share tasks, and (for simplicity) a single pipeline. Assume that the pipeline is synchronous (i.e., clocked) and that the clock or synchronization period is based on the time associated with the maximum stage execution time where that maximum is determined by the tasks assigned to each stage. The execution time for a single flow $j$, on a given stage $k$ is given by:

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

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

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

while 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 $P$ in Equation 4 since $packet\ throughput = 1/P$.

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 (i.e., those that satisfy the constraints discussed above), and select the optimium one. Optimal efficient solutions are unavailable, however, as even restricted versions of this problem have been proven to be NP Hard [10, 11].

Consider first the complexity associated with performing assignments for the single flow case where there are $M$ tasks present and $R$ identical pipeline stages. An upper bound on the number of combinations that must be examined corresponds to the standard problem of calculating the number of ways of placing M identical balls into $(M + R - 1)$ bins. With $N$ flows present each flow having $M_j$ tasks, the number of combinations increases by the product of combinations possible with each flow. Thus:

$$Number\ of\ Assignments \ \leq \ \prod_{j=1}^{N} \binom{M_j + R - 1}{M_j} \tag{5}$$

This results in the worst case count of combinations. This will decrease (sometimes significantly) when there are tasks common between the flows (i.e., shared); however, even with this reduction, complete enumeration is infeasible for analyzing practical systems. Nevertheless, for relatively simple cases, when testing *GreedyPipe*, we have used this approach.

# 3 The *GreedyPipe* Algorithm

We begin this section with a brief review of prior work in this area, next describe the *GreedyPipe* algorithm, and finally review an example of its operation.

## 3.1 Related Work

The assignment problem being considered is similar to a number of problems considered in the literature. For example, there is a long history associated with deterministic job-shop scheduling [11] and these problems have been investigated from a variety of perspectives including integer programming, use of heuristics and related approaches.

Similar problems have also been dealt with in the context of finding compilation techniques for general purpose parallel languages on multiprocessors[12, 13]. Multiprocessor performance in these cases is maximized by identifying potential parallelism and then partitioning the program accordingly to exploit the parallelism. The primary objective of the compilation techniques is to minimize the response time while simultaneously reducing overhead due to inter-process synchronization and communication. Real-time packet scheduling problems have also been considered in the context of network processors [6]. In this case, however, while a non-pipelined parallel processor system was considered, 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 allocation 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 the synchronous clock requirements of the computer pipeline environment. *GreedyPipe*, the heuristic developed, takes these factors into account and thus differs from prior heuristics in this area.

## 3.2 *GreedyPipe*: Basic Ideas

*Greedypipe* is a heuristic based in part on a greedy algorithm and thus gives no guarantee of finding an optimal solution. However, it provides a very good solution quickly and tests

indicate that it finds the optimal solution 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:

$$TotalTime_j = \sum_{i=1}^{M_j} t_{ij} \tag{6}$$

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:

$$Ideal.Delay.per.Stage_j \;=\; \frac{TotalTime_j}{R} \tag{7}$$

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:

$$s_{jk} = \sum_{i=1}^{M_j} X_{ijk} t_{ij} \tag{8}$$

Since, the pipeline is synchronous and is clocked on the basis of the time associated with 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 9.

$$Var_j = \max_{k=1}^{R} \mid s_{jk} \;-\; Ideal.Delay.per.Stage_j \mid \tag{9}$$

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

best assignment now requires Equation 9 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:

$$Var = \max_{j=1}^{N} Var_j = \max_{j=1}^{N} \left\{ \max_{k=1}^{R} \mid s_{jk} \; - \; Ideal.Delay.per.Stage_j \mid \right\} \tag{10}$$

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 $k$, potentially there may be multiple allocations for which the minimized $Var$ has the same value. The heuristic breaks such ties by choosing the assignment, out of the set of tied assignments, that minimizes the following sum:

$$S = \sum_{j=1}^{N} ( \; [j_{jk}] \; - \; Ideal.Delay.per.Stage_j \; ) \tag{11}$$

## 3.3 *Greedypipe*: Overall Algorithm

The overall heuristic begins by calculating the $Ideal.Delay.per.Stage_j$ for each flow $j$ given by Equation 7.

Task-to-stage assignments start with the first pipeline stage. Two sets of tasks satisfying the constraints are selected from each of the flows for allocation to this pipeline stage. These sets are selected so that they correspond to the best currently available bound around the $Ideal.Delay.per.Stage_j$. The first set is chosen so that the variation, $Var_j$, given by Equation 9 (with $R = 1$) is minimized and the inner term ($s_{jk} \; - \; Ideal.Delay.per.Stage_j$) is positive. The second set is chosen similarly, however, in this case the inner term is required to be negative. Additional sets that satisfy the constraints and represent increasingly less stringent bounds may be chosen at the cost of increased complexity and execution time.

For the experiments described later, two positive and two negative assignments were selected for each flow. Thus, for these experiments, there are four best assignments associated

with each flow and thus there are $4^N$ possible combinations of good flow assignments to consider. These combinations are now examined and the two"best" are kept for use in performing task assignments for the next pipeline stage. Tie situations are broken as indicated in the prior section.

Assignments for the next pipeline stage are now considered. The process begins by first calculating new $Ideal.Delay.per.Stage_j$ values based on unallocated tasks and the number of remaining pipeline stages. Next, each of the 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, the next four "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 10) 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 completed. The best of the final stage two assignments is now kept.

Notice that the algorithm has an implicit ordering aspect to it in 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.

## 3.4 A Simple Example

To illustrate operation of *GreedyPipe* we present a simple single flow example that has five tasks and a three stage pipeline. In this example, for simplicity, only a single pair of assignments are generated and passed from one stage to the next. The task times are given in Table 1.

Begin by calculating the $Ideal.Delay.per.Stage_1$ as 4.33 (e.g., (2+4+3+1+3)/3 = 4.33).

| | 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 | 2 | 4 | 3 | 1 | 3 |

Table 1: A Single Flows with five *ordered* tasks

Next consider allocations to stage 1 and calculate $Var_1$. This is shown in complete form in Table 2 although clearly not all calculations need be done.

| Stage 1 Allocation | $Var_1$ | Best Selections |
|---|---|---|
| $T_1$ | $Var_1 \ = \ 4.33 - 2 \ = \ 2.33$ | + |
| $T_1, T_2$ | $Var_1 \ = \ 4.33 - (2 + 4) \ = \ -1.67$ | − |
| $T_1, T_2, T_3$ | $Var_1 \ = \ 4.33 - (2 + 4 + 3) \ = \ -4.67$ | |
| $T_1, T_2, T_3, T_4$ | $Var_1 \ = \ 4.33 - (2 + 4 + 3 + 1) \ = \ -5.67$ | |
| $T_1, T_2, T_3, T_4, T_5$ | $Var_1 \ = \ 4.33 - (2 + 4 + 3 + 1 + 3) \ = \ -8.67$ | |

Table 2: All possible allocations for Stage 1

The best positive selection corresponds to the stage 1 assignment of $T_1$, while the best negative selection corresponds to $T_1, T_2$. Starting with these selections, stage 2 assignments are now considered. Two new $Ideal.Delay.per.Stage_2$ values are now calculated based on stage selections + and − above. They are: $Ideal.Delay.per.Stage_2 = [(4+3+1+3)/2] = 5.5$ or $Ideal.Delay.per.Stage_2 = [(3 + 1 + 3)/2] = 3.5$.

A new set of possible allocations is now calculated. Table 3 shows the four best assignments associated with the starting allocations of stage 1. From these four, the two best are selected for consideration (noted with a *). Notice that the heuristic first breaks the 1.7 tie by employing Equation 11. This leads to the last choice in the table. Since *GreedyPipe* retains two choices at each stage, the upper two choices are now examined. They are tied both in terms of the Max (Equation 10) calculation (2.3) and the use of Equation 11. Thus, for this situation one of them is arbitrarily selected.

The two stage 2 allocation choices are now the starting points for examining the possible assignments for stage 3 as shown in Table 4. Since this is the last stage, all the remaining assignments are dictated by prior stage assignments as shown in the table.

In this case, as Table 4 indicates, there is a single best assignment. The same process is

| Stage 2 Allocation | | | | |
|---|---|---|---|---|
| Stage 1 | Stage 2 | $Var_2$ | Max$[\|Var_1\|, \|Var_2\|]$ | Best Selections |
| $T_1$ | $T_2$ | $Var_2 = 5.5 - 4 = 1.5$ | 2.3 | $*$ |
| $T_1$ | $T_2, T_3$ | $Var_2 = 5.5 - (4+3) = -1.5$ | 2.3 | |
| $T_1, T_2$ | $T_3$ | $Var_2 = 3.5 - 3 = 1.5$ | 1.7 | |
| $T_1, T_2$ | $T_3, T_4$ | $Var_2 = 3.5 - (3+1) = -0.5$ | 1.7 | $*$ |

Table 3: Best allocations for Stage 2

| Stage 3 Allocation | | | |
|---|---|---|---|
| Stage 1 | Stage 2 | Stage 3 | Max. Stage Time |
| $T_1, \quad T = 2$ | $T_2, \quad T = 4$ | $T_3, T_4, T_5, \quad T = 7$ | 7 |
| $T_1, T_2, \quad T = 6$ | $T_3, T_4, \quad T = 4$ | $T_5, \quad T = 3$ | 6 $*$ |

Table 4: Best allocations, forward direction

now repeated starting from stage 3 and task 5 and working in the reverse direction. For this case, the best assignment is the same as obtained in the forward direction. Additionally, it is also the same as obtained when using a complete enumeration approach.

## 3.5 *GreedyPipe* Performance

There are two elements associated with evaluating *GreedyPipe* 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 a time consuming exhaustive search algorithm.

Overall, over a range of randomly generated conditions[3] 98% of the time *GreedyPipe* results are within 15% of the optimal and in no case was the *GreedyPipe* result greater than 25% from the optimal. The results however varied with the number of stages in the pipeline and the percentage of shared tasks associated with different flows. Shared tasks present interesting complications for the heuristic (not dealt with here) and generally result in somewhat higher errors (though within the bounds noted above). For systems containing

[3]Number of stages=[2 to 5]: Number of flows=[1 to 3]; Number of tasks=[6 to 15]; Percentage of sharing=[0 to 100].

1 to 3 flows and a number of tasks per flow equal to three times the number of stages, overall results are as follows:

- For small systems involving 2 or 3 stages; 95% of the time the optimal solution was obtained and 98% of the time the result was within 10% of the optimal.
- For larger systems involving 4 or 5 stages; 72% of the time the optimal solution was obtained and 96% of the time the result was within 10% of the optimal.
- For all systems, when the percentage of task sharing was 25% or less, nearly 100% of the time the result was within 10% of the optimal. When there was no task sharing, more than 99% of the time *GreedyPipe* obtained the optimal result.

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 took on the order of hours versus on the order of seconds with *GreedyPipe*(using a 500MHz Sparc processor). This difference will increase exponentially as the complexity of the system (e.g., number of flows, stages, tasks) increases further.

# 4 Pipeline Design with *GreedyPipe*

In systems with multiple pipelines and flows, such as network processors, 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 (e.g., associated with flows) that have been partitioned into a number of ordered tasks, a designer can select the number of pipeline stages. 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 varying number of stages. This also applies to the number of pipelines to be designed into the system.

- 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 a reduction in cache miss rates. However, when tasks are shared, there is less flexibility in task-to-stage assignments and generally, for a fixed size pipeline, lower overall throughput can be expected. *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. While the experiments differ in their objectives, 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 (floating point numbers).

## 4.1 Number of Pipeline Stages

*GreedyPipe* may be used to determine the effect of pipeline depth on throughput performance. In the network processor context, throughput units are packets per second. This is illustrated

in Figure 2 where the results for a system with 6 flows and 20 tasks per flow is shown with none of the flows sharing tasks.

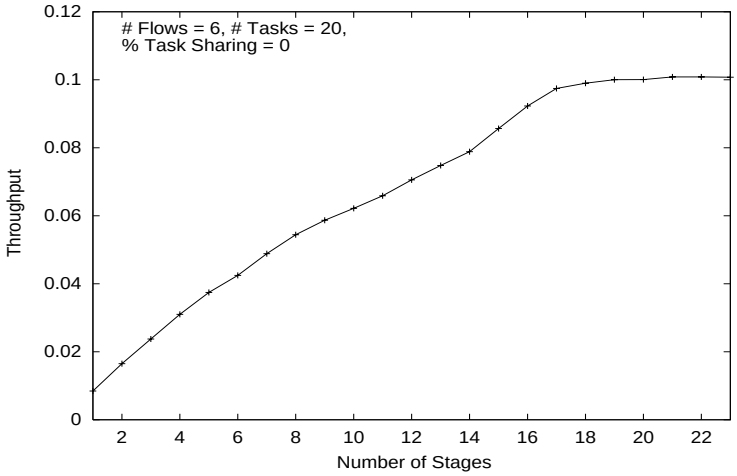


Figure 2: Throughput(packets/sec) vs Number of Pipeline Stages

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 about sixteen stages. After that, it is more difficult to evenly distribute the tasks over the stages and the throughput asymptotically approaches its maximum of about 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 a value close to 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

In Section 2.1, an assignment constraint was introduced such that a task may only be assigned to one processor. When tasks are common (i.e., shared) between flows, this constraint leads to interdependence between the flows. Task sharing may result in better memory utilization and lower instruction cache miss rates, however, it also restricts the number of assignment options that may be considered and thus potentially reduces the throughput that is achievable.

Experiments were conducted for the case of 6 flows, 20 tasks per flow and a single 8 stage pipeline. To see the effect of increasing the number of shared tasks between flows, 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 shown in 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.

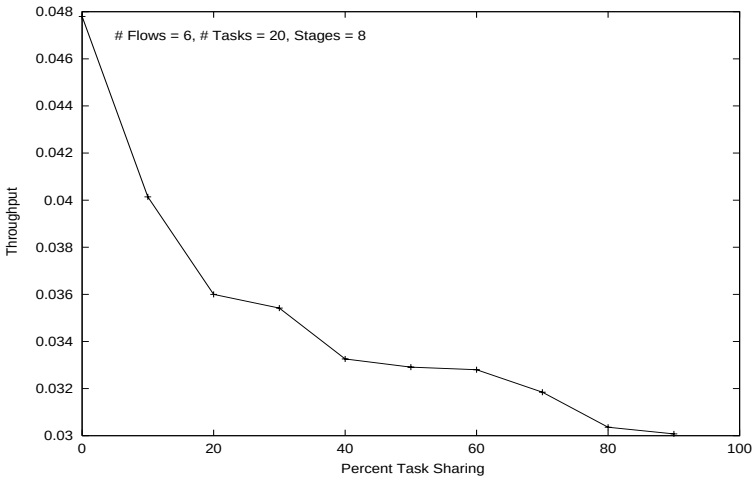


Figure 3: Throughput(packets/sec) vs Percent Shared Tasks

## 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 having greater flexibility in assigning the tasks to the hardware pipeline and this usually results in higher throughput. Additionally, having more tasks for a given application means that longer pipelines can be effectively used which again often 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 nontrivial development

costs associated with this approach. 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, a single 5 stage pipeline, and no task sharing. Initially the number of tasks is set at 11 tasks per flow. For the lower curve, after the first *GreedyPipe* task-to-stage assignment, the single longest task in each flow is divided in half creating two smaller tasks. This is repeated for each successive "Partition Cycle" with the final cycle in the figure corresponding to the assignment of 20 tasks per flow. For the upper curve, the two largest tasks are divided in half at each cycle. Thus, for this latter case, the number of tasks varies from 11 to 29 tasks.

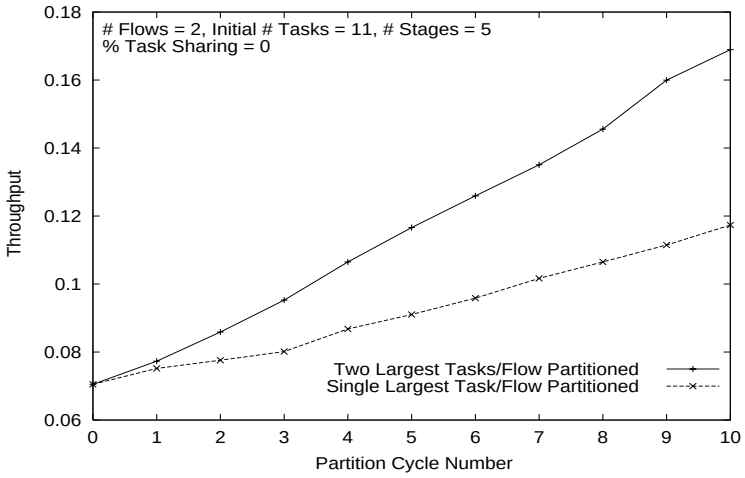


Figure 4: Throughput(packets/sec) vs Task Partitioning

Clearly, dividing the application into more tasks is beneficial since there are more opportunities for improved task assignments. Also, as the set of longest tasks to be divided is increased, potential bottlenecks arising from the longest tasks are removed and resulting assignments further increase throughput. Thus, *GreedyPipe* permits the designer to quickly

determine the potential benefit associated with spending more time on algorithm partitioning.

# 5 Summary and Conclusions

This paper has presented *GreedyPipe*, a heuristic approach for quickly performing very good task-to-pipeline stage assignments in a multiple flow, multiple pipeline environment. Obtaining optimal assignments is an NP hard problem and, for assignment problems of interest, exhaustive search techniques can take many hours to find an optimal assignment. *GreedyPipe* obtains very good assignments within seconds and thus can be used to effectively explore alternative design options.

While *GreedyPipe* can be used in obtaining throughput performance in a variety of pipelining environments, its development was motivated by the growing importance of embedded processor systems that are based on having multiple pipelines on a single chip. This sort of design is used extensively in current network processors and future work will focus on applying *GreedyPipe* to applications associated with such systems.

In addition to presenting details associated with the heuristic, its design and performance, the paper also illustrates how *GreedyPipe* can be employed as a design tool in situations where the effects of pipeline depth, task sharing and task partitioning are to be explored as part of the design process.

# References


[1] IBM Corp., "IBM Power Network Processors," 2000.

[2] Intel Corp., "Intel IXP 2800 Network Processor," 2000.

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

[4] M. Franklin and T.Wolf, "A Network Processor Performance and Design Model with Benchmark Parameterization," in *Proc. 1st Workshop on Network Processors, in conjunction with*

*8th Inter. Symp. on High Performance Computer Architecture (HPCA-8), Cambridge, MA.*, Feb 2002.

[5] M. Franklin and T.Wolf, "Power Consideration in Network Processor Design," in *Proc. 2nd Workshop on Network Processors, in conjunction with 9th Inter. Symp. on High Performance Computer Architecture (HPCA-9), Cambridge, MA.*, Feb 2003.

[6] 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.

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

[8] Wei-Je Huang and N. Saxena and E. J. McCluskey, "A Reliable LZ Data Compressor on Reconfigurable Coprocessors," in *IEEE Symp. on Field-Programmable Custom Computing Machines*, (Napa Valley, California), pp. 249–258, April 2000.

[9] 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.

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

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

[12] 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.

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