Automatically Partitioning Packet Processing Applications for Pipelined Architectures

Jinquan Dai, Bo Huang, Long Li
Intel China Software Center
22nd Floor, ShanghaiMart Tower
No. 2299 Yan'an Road (West), Shanghai, 200336, PRC
86-21-52574545 ext. {1615, 1373, 1647}
{jason.dai, bo.huang, paul.li}@intel.com

Luddy Harrison
Department of Computer Science
Univ. of Illinois at Urbana-Champaign
201 N. Goodwin, Urbana, IL 61801
1-217-244-2882
luddy@uiuc.edu

Abstract

Modern network processors employs parallel processing engines (PEs) to keep up with explosive internet packet processing demands. Most network processors further allow processing engines to be organized in a pipelined fashion to enable higher processing throughput and flexibility. In this paper, we present a novel program transformation technique to exploit parallel and pipelined computing power of modern network processors. Our proposed method automatically partitions a sequential packet processing application into coordinated pipelined parallel subtasks which can be naturally mapped to contemporary high-performance network processors. Our transformation technique ensures that packet processing tasks are balanced among pipeline stages and that data transmission between pipeline stages is minimized. We have implemented the proposed transformation method in an auto-partitioning C compiler product for Intel Network Processors. Experimental results show that our method provides impressive speed up for the commonly used NPF IPv4 forwarding and IP forwarding benchmarks. For a 9-stage pipeline, our auto-partitioning C compiler obtained more than 4X speedup for the IPv4 forwarding PPS and the IP forwarding PPS (for both the IPv4 traffic and IPv6 traffic).

Descriptors D.3.4 [**Programming Languages**]: Processors – *compilers, optimization*.

General Terms Algorithms, Performance, Design.

Keywords Network Processor, Pipelining Transformation, Program Partition, Live-Set Transmission, Parallel, Packet Processing

1. Introduction

Internet traffic has grown at an explosive rate. The increasing amounts of services offered on the internet have continually pushed the network bandwidth requirement to newer heights [1]. Advances in microprocessor technology helped pave the way for the development of Network Processors (NPs) [2][3][4][5][6], which are designed specifically to meet the requirements of next generation network equipments.

In order to address the tremendous speed challenges of network processing, modern Network Processors generally have a parallel multiprocessor architecture with multiple processing elements (PEs) on a single chip. The PEs often are controlled by a general purpose processor and supported by other reconfigurable logic elements. In many NPs the processing elements can also be organized as a pipeline, providing computing and algorithmic flexibility. In this case, a packet processing application can be partitioned into several pipeline stages, with each processing element containing one pipeline stage. The packet processing application as a whole can be accelerated by a factor of up to the number of pipeline stages [2][3][4][5][6]. The Intel IXA NPU family of network processors (IXP), for instance, contains multiple MicroEngines (MEs) which can be deployed either as a pipeline or as a pool of homogeneous processors operating on distinct packets [2].

The unique challenge of network processing is to guarantee and sustain the throughput of packet processing for worst-case traffic. That is, each network application has performance requirements that have to be statically guaranteed. Therefore a static compiler, rather than a dynamic or runtime approach, is preferred in this area. In order to exploit the underlying parallel and pipelined architecture for higher performance, existing compilers for NPs, (e.g., the Intel® MicroEngine C compiler), usually adopts the paradigm of parallel programming for network applications, as practiced in the scientific computing community. This requires that the programmers manually partition the application into sub-tasks, manage the synchronization and communication among different sub-tasks, and map them onto a multiprocessor system explicitly. Unfortunately, such a parallel programming paradigm is not intuitive and not familiar to most programmers.

On the other hand, network applications are most naturally expressed in a sequential way. When a new packet arrives, the application performs a series of tasks (e.g., receipt of the packet, routing table look-up, and enqueuing) on the packet. Consequently, there is a large gap between the parallel programming paradigm on NPs and the sequential semantics of network applications.

Prior work on advanced tools for mapping packet processing application onto processing elements can be found in [9][10][11]. In this paper, we propose a novel approach that automatically transforms sequential packet processing applications into pipelined forms for execution on pipelined packet processing architectures. The proposed algorithm automatically partitions a sequential packet processing application into chained pipeline stages and ensures that 1) the packet processing tasks are balanced among pipeline stages and 2) the data transmission between pipeline stages is minimized.

PLDI'05, June 12–15, 2005, Chicago, Illinois, USA.

The rest of the paper is organized as follows. In section 2, we provide a brief overview of the IXP architecture [2] and the auto-partitioning programming model used by the input programs we consider [7]. In section 3 we present the algorithms for automatic pipeline transformation and live set transmission minimization. We present experimental results in section 4 and cover related work in section 5. Finally, we draw conclusions and describe future work in section 6.

2. The IXP architecture for packet processing

The IXP network processor architecture [2] is designed to process packets at high rates, i.e., where inter-arrival times between packets may be less than a single memory access latency. To keep up with such high arrival rates, IXP presents a novel architecture consisting of one core processor based on the Intel® XScale and a collection of multithreaded packet processing engines with an explicit memory hierarchy. For example, the IXP2800 contains sixteen processing engines (PE in Figure 1). The processing engines have a non-traditional, highly parallel architecture. Ordinarily, XScale runs the control plane code and the packet engines run the fast path data plane code.

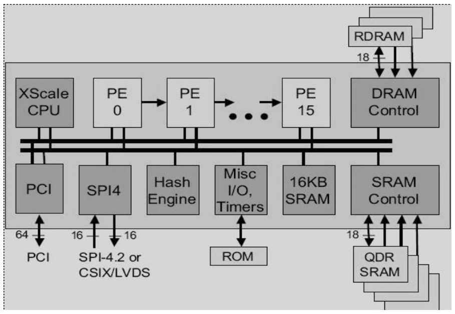


Figure 1. Intel IXP2800 block diagram

2.1 Pipelined processing engines

Each processing engine in an IXP has eight hardware threads, with zero-overhead context switching between them. Although each packet engine is an independent processor with its own register file and 32-bit ALU, several packet engines can work together to form an engine level pipeline. A packet processing application can be mapped onto such a pipeline by dividing it into successive stages, each containing a portion of the packet processing tasks.

Data transmission between successive pipeline stages is accomplished on IXP using *ring buffers.* The IXP provides two flavors of hardware-supported rings: 1) a nearest neighbor (NN) ring, a register-based ring that can deliver words between packet engines in just a few cycles, and 2) a scratch ring, which is implemented in static memory and takes on the order of a hundred cycles to perform an enqueue/dequeue operation.

2.2 Packet processing stages

In order to reduce the programming effort required to develop a fast-path data plane packet processing application for IXP, an auto-partitioning C compiler product [7] is being developed by Intel. The goal of this compiler is to automatically partition packet processing applications onto multiple processing engines and threads, thus allowing programmers to develop applications using sequential semantics and without excessive concern for the details of the machine's organization.

The auto-partitioning C compiler requires that the input program be developed using the auto-partitioning programming model, in which the data plane packet processing application is expressed as a set of sequential C programs called *packet processing stages (PPSes).* This model corresponds closely to the *communicating sequential processes (CSP)* model of computation [8] in which independent sequential programs run concurrently and communicate via queues. A PPS is a logical entity written using hardware-independent sequential C constructs and libraries, and is not bound by the programmer to a specific number of compute elements (processing engines, threads etc.) on the IXP. Each PPS contains an infinite loop, also called a *PPS loop*, which performs the packet processing indefinitely. The primary mechanism by which PPSes communicate with one another is a *pipe* which is an abstract, unidirectional communication channel, i.e., a queue. Like a PPS, a pipe is also a logical entity that is not bound by the programmer to a specific physical communication channel (NN rings, scratch rings, SRAM rings) on the IXP. PPSes can also communicate through variables in shared memory [7].

The auto-partitioning C compiler automatically explores how (e.g., pipelining vs. multiprocessing) each PPS is paralleled and how many PEs (e.g., number of pipeline stages or multiprocessing stages) each PPS is mapped onto, and selects one compilation result based on a static evaluation of the performance and the performance requirements of the application. The pipelining transformation presented in this paper is a fundamental algorithm in the compiler; on the other hand, how the exploration and multiprocessing are performed is beyond the scope of this paper.

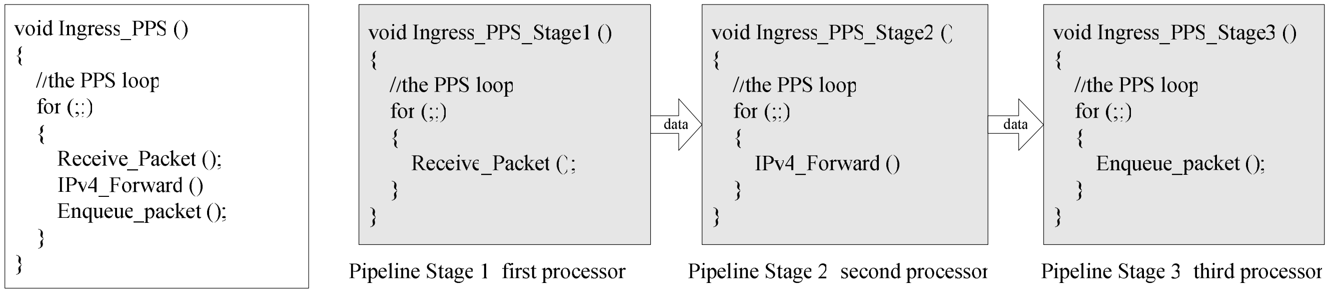
The expression of a packet processing application as a set of communicating PPSes represents the logical partitioning of the application into concurrently executing processes. This is done by the programmer, who may write as few or as many PPSes as seems natural for the application at hand. The algorithm for pipeline decomposition presented in this paper operates on a single PPS, and represents the decomposition of the PPS into a physical form that is appropriate for high performance execution on the IXP.

3. Automatic pipelining transformations

In this section, we describe a novel algorithm for transforming a sequential PPS into pipelined parallel form, as shown in Figure 2a and 2b. By this transformation two or more processing engines in an NP are organized as a pipeline where each stage contains a portion of the original PPS loop. Data that are alive at the boundary between one stage and the next are communicated from the earlier stage to the later, as shown in Figure 3.

3.1 The framework of pipelining transformation

A *cut* is a set of control flow points that divide the PPS loop body into two pieces. If the PPS is to be partitioned into *D* stages, then *D–1* cuts must be selected, and the cuts must be non-overlapping. That is, each node in the control flow graph must lie in one pipeline stage after all the cuts are applied. For better performance, the selected cuts need to meet the following criteria.



```
void Ingress_PPS ()
{
   //the PPS loop
   for (;;)
   {
      Receive_Packet ();
      IPv4_Forward ()
      Enqueue_packet ();
   }
}
```

```
void Ingress_PPS_Stage1 ()
{
   //the PPS loop
   for (;;)
   {
      Receive_Packet ();
   }
}
```

Pipeline Stage 1 first processor

```
void Ingress_PPS_Stage2 ()
{
   //the PPS loop
   for (;;)
   {
      IPv4_Forward ()
   }
}
```

Pipeline Stage 2 second processor

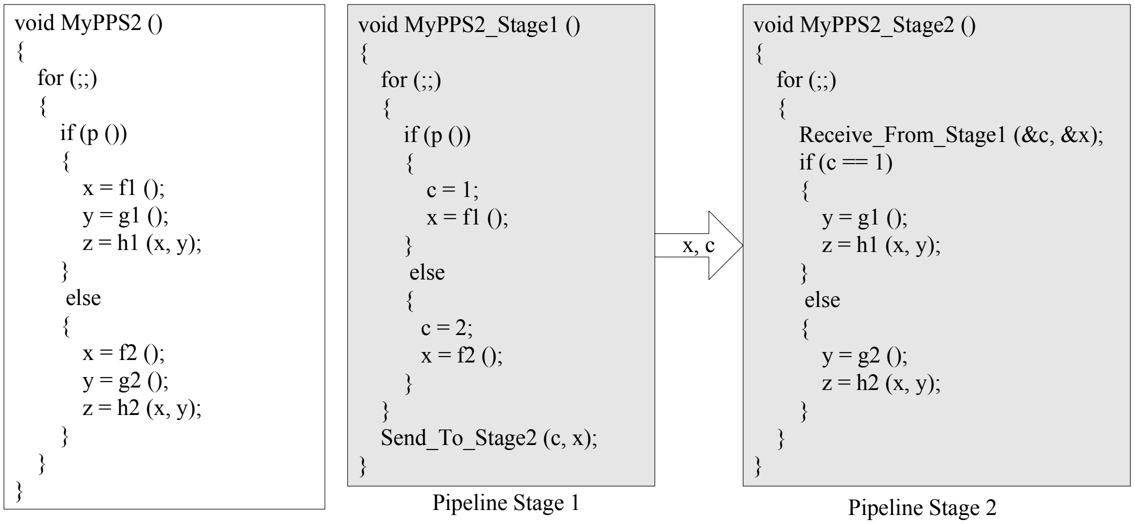
```
void Ingress_PPS_Stage3 ()
{
   //the PPS loop
   for (;;)
   {
      Enqueue_packet ();
   }
}
```

Pipeline Stage 3 third processor

(a) The original sequential PPS

(b) The tranformed PPSes that are pipelined

Figure 2. The ingress of IPv4 forwarding processing application



```
void MyPPS2 ()
{
   for (;;)
   {
      if (p ())
      {
         x = f1 ();
         y = g1 ();
         z = h1 (x, y);
      }
      else
      {
         x = f2 ();
         y = g2 ();
         z = h2 (x, y);
      }
   }
}
```

```
void MyPPS2_Stage1 ()
{
   for (;;)
   {
      if (p ())
      {
         c = 1;
         x = f1 ();
      }
      else
      {
         c = 2;
         x = f2 ();
      }
   }
   Send_To_Stage2 (c, x);
}
```

Pipeline Stage 1

```
void MyPPS2_Stage2 ()
{
   for (;;)
   {
      Receive_From_Stage1 (&c, &x);
      if (c == 1)
      {
         y = g1 ();
         z = h1 (x, y);
      }
      else
      {
         y = g2 ();
         z = h2 (x, y);
      }
   }
}
```

Pipeline Stage 2

(a) The original sequential PPS

(b) The tranformed PPSes that are pipelined

Figure 3. Proper transmissions of the live set cross cuts

1. *No dependence from later stages to earlier ones*
   Any dependence from a later stage to an earlier one is necessarily PPS loop-carried. Should such dependences exist, feedback paths from later stages to earlier ones are required to ensure that the earlier stage (of a later iteration) stalls until the later stage (of an earlier iteration) satisfies the dependence. Backward-flowing synchronization is complex and awkward to implement on the IXP. We have chosen to prohibit such dependences in our choice of cuts, in favor of simple, unidirectional communication and synchronization between stages.

2. *Minimization of live set*
   After cutting the PPS into two stages, data that are alive at the cut (roughly speaking, the contents of live registers) must be transmitted across the cut, so that the downstream pipeline stage may begin executing in the proper context. In addition, some control flow information must be transmitted over the cut so that the downstream stage may begin executing at the right program point. We call this data collectively the *live set* (see variables *x* and *c* in Figure 3).

3. *Balance of packet processing tasks*
   The performance of the pipelined computation as a whole will be no better than the *slowest* of the pipeline stages. For this reason it is desirable to balance the original packet processing tasks among the stages as evenly as possible.

It is natural to model the problem of selecting cuts as a network flow problem, in which the weight of an edge represents the cost of transferring the live set between pipeline stages if the edge is cut. As a result, the selection of cuts is reduced to finding minimum cuts of the flow network that result in balanced instruction counts among pipeline stages. Consequently, the overall framework of pipelining transformation consists of *construction of a proper flow network model*, *selection of cuts on the flow network*, and *realization of pipeline stages*.

3.2 Construction of the flow network model

A proper flow network model should help us avoid dependences from the downstream nodes of the cut to upstream ones, and should model correctly the cost of transmission of the live set. The flow network is constructed from the single static assignment (SSA) form of the program and the dependence graph of the program. The flowchart of the construction process is shown in Figure 4, and the steps are described in detail in subsequent sections.

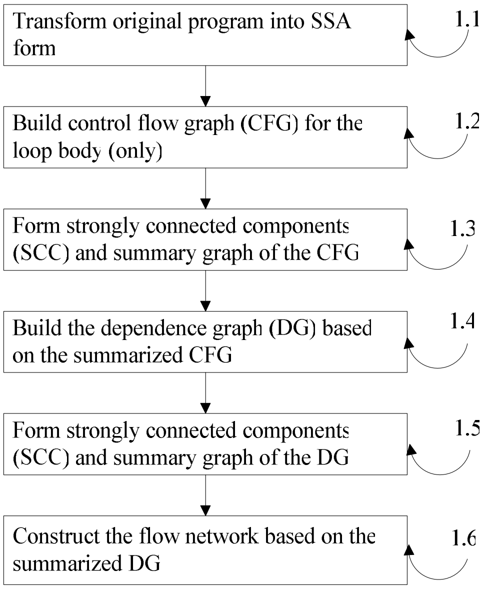

Transform original program into SSA form
1.1
Build control flow graph (CFG) for the loop body (only)
1.2
Form strongly connected components (SCC) and summary graph of the CFG
1.3
Build the dependence graph (DG) based on the summarized CFG
1.4
Form strongly connected components (SCC) and summary graph of the DG
1.5
Construct the flow network based on the summarized DG
1.6

Figure 4. The flow chart for the construction of the flow network model

3.2.1 Elimination of PPS loop carried dependence

To eliminate control dependences from later stages to earlier ones, the pipelining transformation should not split any strongly connected component (SCC) of the control flow graph (CFG) across pipeline stages. That is, each SCC should belong in its entirety to one pipeline stage after the transformation. Step 1.3 therefore forms SCCs for the CFG and builds the associated summarized graph (in which SCCs are reduced to a single node), and step 1.4 constructs the dependence graph (DG) based on the summarized graph.

To eliminate data dependences from later stages to earlier ones, step 1.4 includes PPS-loop-carried flow dependence as well as non-loop-carried data and control dependence in the DG; consequently, the sources and the sinks of the PPS-loop-carried flow dependence are in the same SCC of the DG. Step 1.5 then forms SCCs for the DG, and the pipelining transformation considers only cuts that place a whole SCC of the DG on one or another side of each cut.

3.2.2 Cost of live set transmission

The flow network makes explicit the flow of values (both variables and control objects) in the program, so that the cost of the live set transmission can be modeled appropriately. The flow network is constructed based on the summarized DG in step 1.6, which is shown in detail in Figure 5.

In addition to the unique *source* and *sink* nodes (step 1.6.1) and *program nodes* that contains instructions (step 1.6.2), *variable node*s and *control nodes* are introduced in the flow network for each object that may be included in the live set (step 1.6.3 and 1.6.4). After the SSA transformation in step 1.1, every variable has only one definition point, and hence has only one *definition edge* (step 1.6.5); and likewise for control nodes (step 1.6.7).

Consequently, the *weight* (or *capacity*) associated with the definition edges (*VCost* for variables and *CCost* for control object) models the cost of transmitting the associated variable or control object if that edge is cut. Its value depends on the underlying architecture of the NPs; since the static guarantee of performance is required, the architecture of the NPs (e.g., IXP) is very predictable and those costs can be statically determined.

In addition, the weight of edges going out of the source and coming into sink are set to 0, as cutting such an edge will not incur any transmission of live set. All the other edges have infinite weights so that they are not eligible for cutting.

3.3 Selection of cuts in the flow network

To cut the PPS into *D* (*the pipelining degree*) stages, the transformation applies *D–1* successive cuts to the PPS such that each cut is a balanced minimum cost cut; the overall framework is shown in Figure 6.

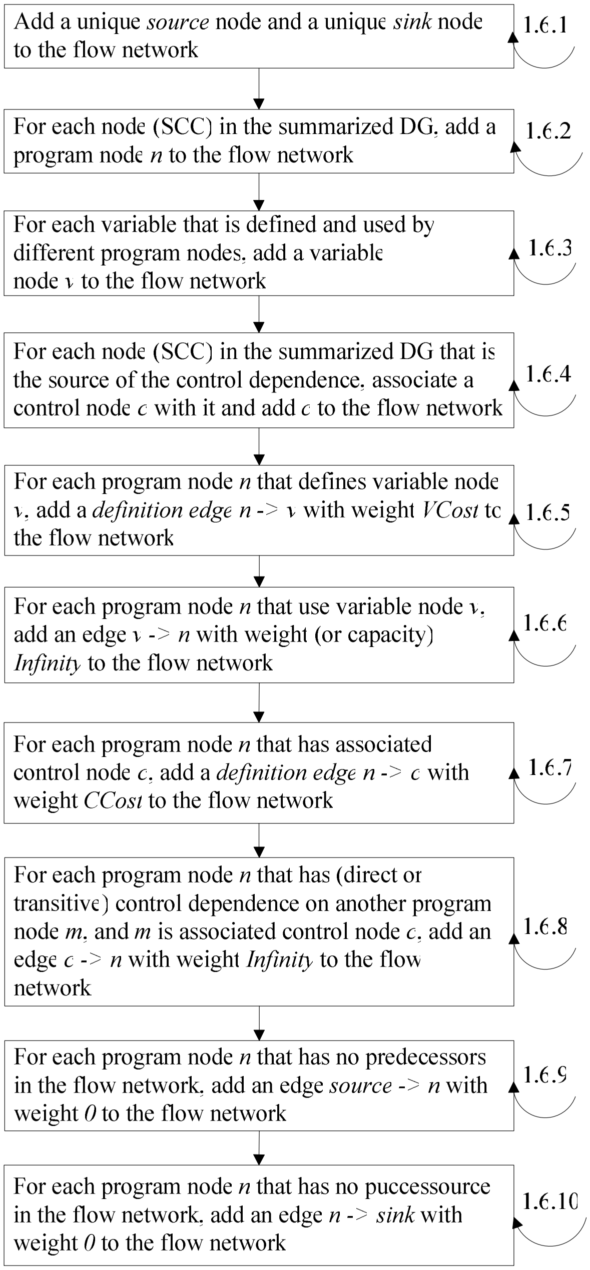

Add a unique *source* node and a unique *sink* node to the flow network
1.6.1
For each node (SCC) in the summarized DG, add a program node *n* to the flow network
1.6.2
For each variable that is defined and used by different program nodes, add a variable node *v* to the flow network
1.6.3
For each node (SCC) in the summarized DG that is the source of the control dependence, associate a control node *c* with it and add *c* to the flow network
1.6.4
For each program node *n* that defines variable node *v*, add a *definition edge n* -> *v* with weight *VCost* to the flow network
1.6.5
For each program node *n* that use variable node *v*, add an edge *v* -> *n* with weight (or capacity) *Infinity* to the flow network
1.6.6
For each program node *n* that has associated control node *c*, add a *definition edge n* -> *c* with weight *CCost* to the flow network
1.6.7
For each program node *n* that has (direct or transitive) control dependence on another program node *m*, and *m* is associated control node *c*, add an edge *c* -> *n* with weight *Infinity* to the flow network
1.6.8
For each program node *n* that has no predecessors in the flow network, add an edge *source* -> *n* with weight *0* to the flow network
1.6.9
For each program node *n* that has no puccessource in the flow network, add an edge *n* -> *sink* with weight *0* to the flow network
1.6.10

Figure 5. The detailed flow chart for building the flow network

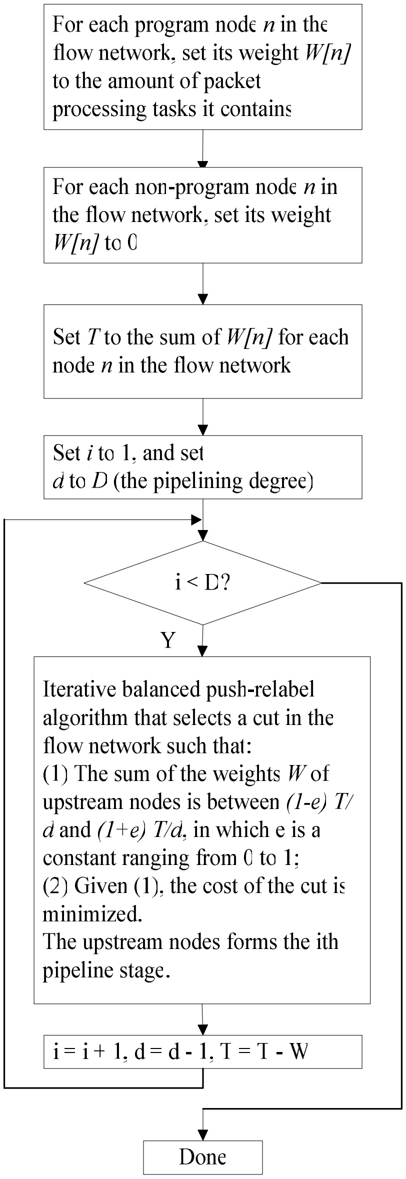

For each program node n in the flow network, set its weight W[n] to the amount of packet processing tasks it contains
For each non-program node n in the flow network, set its weight W[n] to 0
Set T to the sum of W[n] for each node n in the flow network
Set i to 1, and set d to D (the pipelining degree)
i < D?
Y
Iterative balanced push-relabel algorithm that selects a cut in the flow network such that:
(1) The sum of the weights W of upstream nodes is between (1-e) T/d and (1+e) T/d, in which e is a constant ranging from 0 to 1;
(2) Given (1), the cost of the cut is minimized.
The upstream nodes forms the ith pipeline stage.
i = i + 1, d = d - 1, T = T - W
Done

Figure 6. The flow chart for selection of cuts in the flow network

Our algorithm for selecting a balanced minimum cost cut is based on the *iterative balanced push-relabel algorithm* [12]. (It is adapted from [13], and its flow chart is shown in Figure 7). Given a flow network $N = (V, E)$, a weight function W for each node in V, this heuristic applies the push-relabel algorithm iteratively on the flow network N until it finds a cut C which partitions V into X and $V–X$, such that $(1 - e) \cdot W(V) / d \leq W(X) \leq (1 + e) \cdot W(V) / d$, where d is the *balance degree*, and e (a small constant between 0 and 1) is the *balance variance*.

The weight function W of each node models how the placement of the node affects the overall balance of the packet processing tasks; it is flexible and can model various factors (e.g., instruction count, instruction latency, hardware resources, or combinations thereof). In our implementation, instruction count is used because the latency is optimized and hidden through multi-threading, and because code size reduction is an important secondary goal. In the future, other factors might be included for consideration.

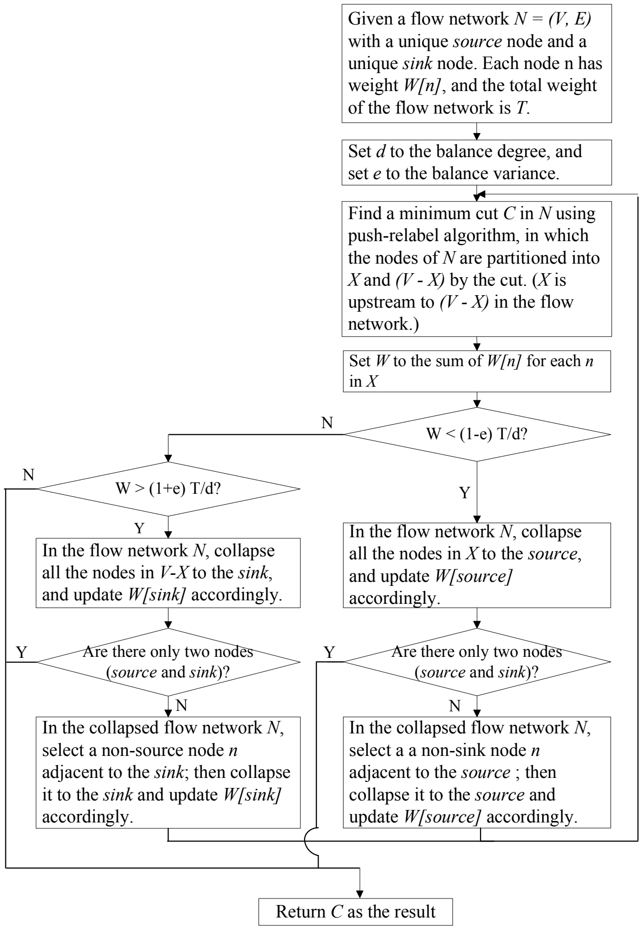
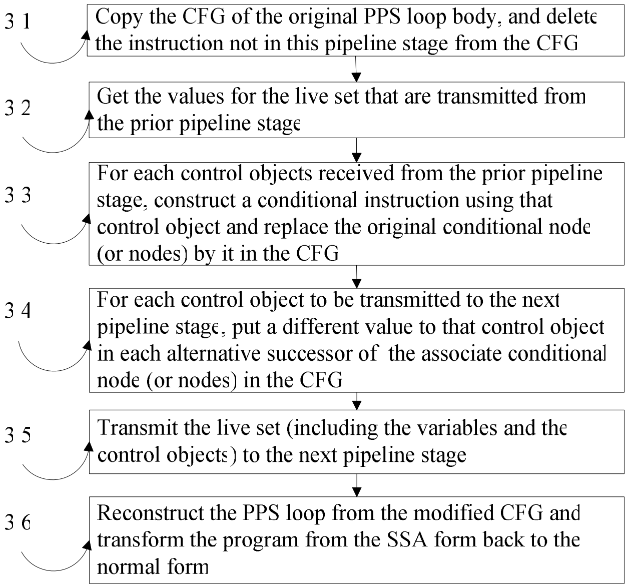

Given a flow network N = (V, E) with a unique source node and a unique sink node. Each node n has weight W[n], and the total weight of the flow network is T.
Set d to the balance degree, and set e to the balance variance.
Find a minimum cut C in N using push-relabel algorithm, in which the nodes of N are partitioned into X and (V - X) by the cut. (X is upstream to (V - X) in the flow network.)
Set W to the sum of W[n] for each n in X
W < (1-e) T/d?
N
Y
W > (1+e) T/d?
N
Y
In the flow network N, collapse all the nodes in V-X to the sink, and update W[sink] accordingly.
In the flow network N, collapse all the nodes in X to the source, and update W[source] accordingly.
Are there only two nodes (source and sink)?
Y
N
Are there only two nodes (source and sink)?
Y
N
In the collapsed flow network N, select a non-source node n adjacent to the sink; then collapse it to the sink and update W[sink] accordingly.
In the collapsed flow network N, select a a non-sink node n adjacent to the source ; then collapse it to the source and update W[source] accordingly.
Return C as the result

Figure 7. The flow chart for the selection of the cuts in the flow network

The balance variance e reflects the tradeoff between the balance and the cost of the cut (in terms of live set transmission). If it is close to 0, the algorithm prefers a balanced cut over one with a smaller cost; on the other hand, if it is close to 1, minimization of the cost is regarded as more important. Experiments may be needed to determine the right value of the balance variance; for instance, its value is set to *1/16* in out implementation, as a result of experimentation and tuning of real world applications.

In addition, an efficient implementation of the heuristic need not run the push-relabel algorithm from scratch in every iteration. Instead, it can be computed incrementally as follows.

- Find the initial minimum cut for the flow network using plain push-relabel algorithm.
- After nodes are collapsed to the source or sink, find the updated minimum cut using the push-relabel algorithm with the following initial states of *pre-flow*, *label*, and *excess* (see [12] for the definition of these variables).
    - Set the pre-flow of all the edges going out of the source to their capacities, and update their excesses accordingly. Leave the pre-flow of other edges unchanged.
    - Set the label of the source to the new number of nodes.
    - If nodes are collapsed to the source, leave the labels of other nodes unchanged. Otherwise, set them to 0.



3.1 Copy the CFG of the original PPS loop body, and delete the instruction not in this pipeline stage from the CFG

3.2 Get the values for the live set that are transmitted from the prior pipeline stage

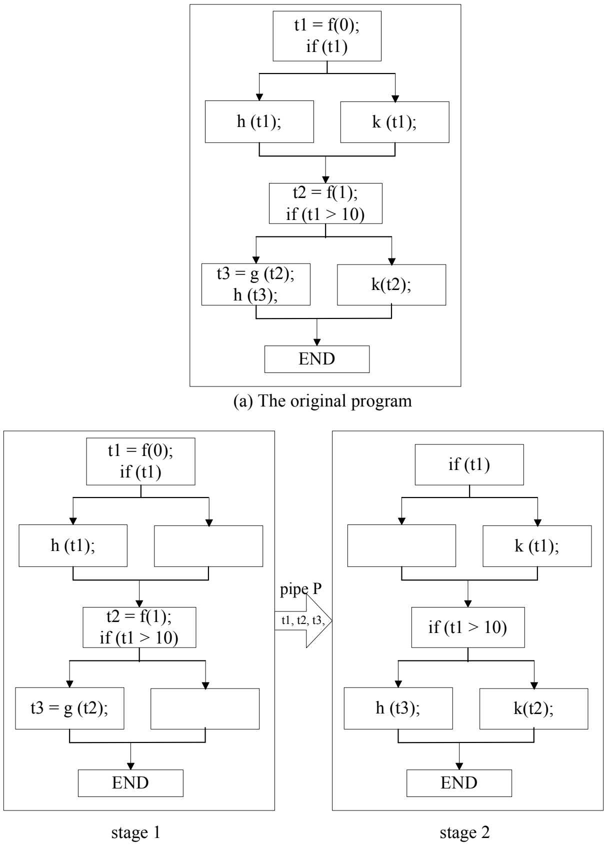
3.3 For each control objects received from the prior pipeline stage, construct a conditional instruction using that control object and replace the original conditional node (or nodes) by it in the CFG

3.4 For each control object to be transmitted to the next pipeline stage, put a different value to that control object in each alternative successor of the associate conditional node (or nodes) in the CFG

3.5 Transmit the live set (including the variables and the control objects) to the next pipeline stage

3.6 Reconstruct the PPS loop from the modified CFG and transform the program from the SSA form back to the normal form

Figure 8. The flow chart for realization of a pipeline stage



(a) The original program

(b) The pipelined program

Figure 9. Example of live set transmission

3.4 Realization of pipeline stages

The realization of a pipeline stage involves proper transmission of the live set across the cut, and the reconstruction of the control flow of the stage. The flow chart of the realization algorithm is shown in Figure 8; the pipelining transformation applies this process to every pipeline stage.

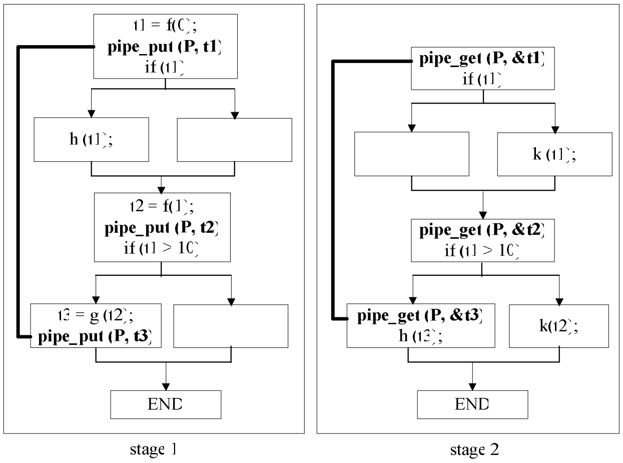


Figure 10 Conditionalized live set transmissions

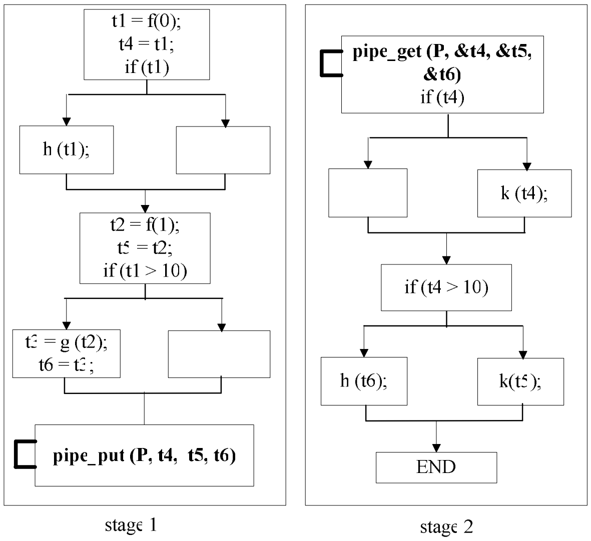


Figure 11. Naively unified live set transmissions

3.4.1 Live set transmission

As shown in Figure 9, for proper transfer of the data and control flow between neighboring stages, the live set needs to be properly transmitted across the cut, through the inter-processor communication channels provided by the NP. A problem that arises is that the live set at one control-flow point (i.e., one edge in the cut) may be different from the live set at a different point.

One resolution is to conditionalize the transmission of every object in the live set, as shown in Figure 10. However, if the pipeline stages are to be multi-threaded later, the transmission of the live set has to be ordered and synchronized across multiple threads, due to the global resource (pipe) used. With the conditionalized transmissions, the critical section around the pipe operations can be very large (as suggested by the bold lines in Figure 10), and consequently the performance of the application is greatly impacted.

Instead, a unified transmission can be used, in which all variables that at any edge in the cut are transmitted with a single aggregate (*unified*) transmission. In this case the critical section around pipe operations is much smaller (as suggested by the bold lines in Figure 11).

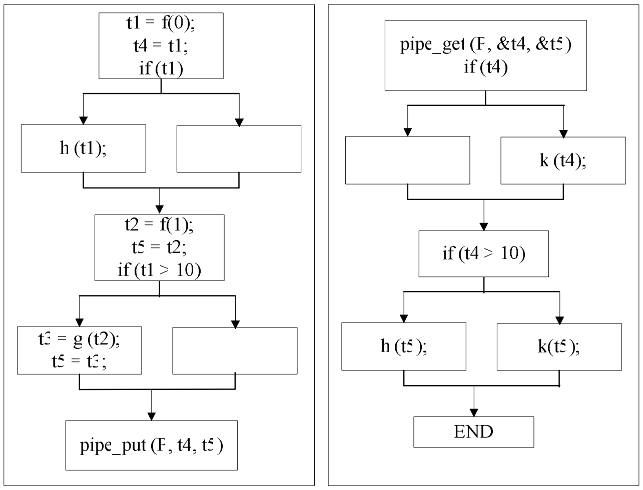


Figure 12. Minimized unified live set transmission

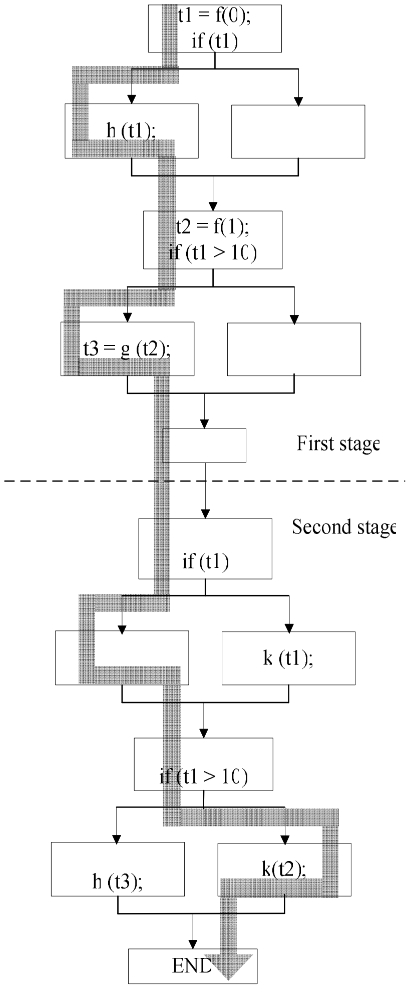
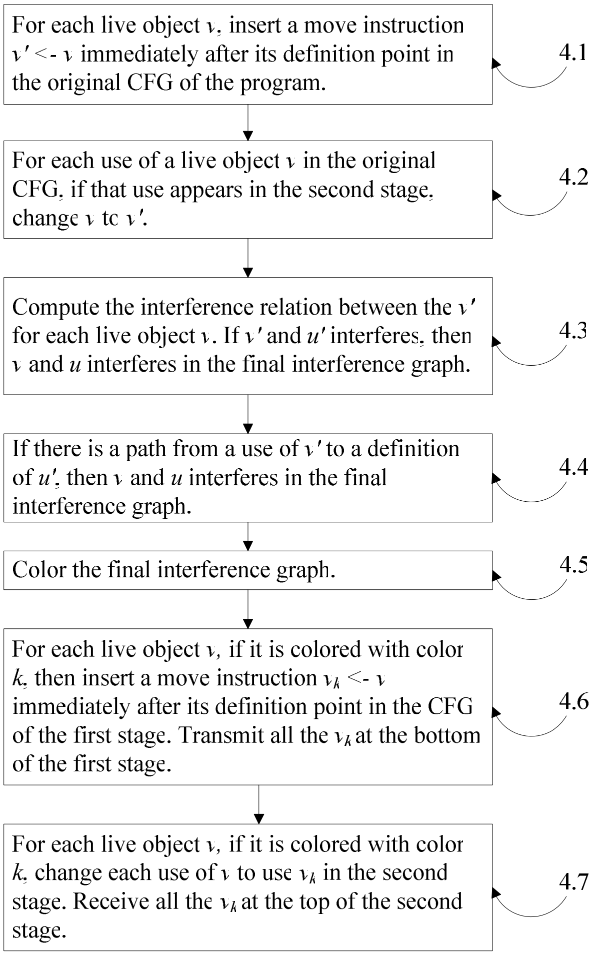


Figure 13. Concatenated flow graphs of the two stages

However, a naïve implementation of the unified transmission, as shown in Figure 11, can transmit more objects than necessary, because two objects in the live set may not be alive at the cut simultaneously (for instance, *t2* and *t3* in Figure 9(b)) and hence only one of them need be transmitted. Ideally, the live set should be packed such that if several objects are never alive at a cut simultaneously (i.e., they do not interfere with each other in the pipelined program), only one of them is transmitted, as illustrated in Figure 12.

Packing the live set can be achieved by first computing an interference relation between objects in the live set, and then coloring each object to a temporary for transmission. If the interference relation is simply computed over the back-to-back concatenated CFGs of the two stages, as shown in Figure 13, false interference edges may be present (such as the interference between *t2* and *t3* in Figure 13), because some paths in the concatenated control flow graph (such as the one shown in Figure 13) can never be executed in reality and should be excluded when computing the interference.

The flow chart for computing the desired interference is shown in Figure 14. In steps 4.1 and 4.2, the original program is rendered such that definitions of the live objects in the current stage and their use in the later stages are made explicit.



For each live object *v*, insert a move instruction *v'* <- *v* immediately after its definition point in the original CFG of the program. 4.1

For each use of a live object *v* in the original CFG, if that use appears in the second stage, change *v* to *v'*. 4.2

Compute the interference relation between the *v'* for each live object *v*. If *v'* and *u'* interferes, then *v* and *u* interferes in the final interference graph. 4.3

If there is a path from a use of *v'* to a definition of *u'*, then *v* and *u* interferes in the final interference graph. 4.4

Color the final interference graph. 4.5

For each live object *v*, if it is colored with color *k*, then insert a move instruction v_k <- *v* immediately after its definition point in the CFG of the first stage. Transmit all the v_k at the bottom of the first stage. 4.6

For each live object *v*, if it is colored with color *k*, change each use of *v* to use v_k in the second stage. Receive all the v_k at the top of the second stage. 4.7

Figure 14. The flow chart for the minimum and unified live set transmission

The steps 4.3 and 4.4 collectively compute the correct interference relation between the live objects, over the back-to-back concatenated CFGs of the two stages with impossible paths excluded. This is because the live object *v* is alive when the live object *u* is defined in the concatenated CFG (excluding impossible paths), if and only if

1) There is a path V1→U1→W1→V2→U2→W2 in the concatenated CFG, where V1 and U1 are the definition points of *v* and *u* in the first stage respectively, V2 and U2 are the counterparts of V1 and U1 in the second stage, W2 is a use of v in the second stage, and W1 is the counterpart of W2 in the first stage (see Figure 15). In this case, *v'* and *u'* interfere in the rendered program and this is computed by step 4.3.
2) There is a path V1→W1→U1→V2→W2→U2 in the concatenated CFG, where V1 and U1 are the definition points of *v* and *u* in the first stage respectively, V2 and U2 are the counterparts of V1 and U1 in the second stage respectively, W2 is a use of v in the second stage, and W1 is the counterpart of W2 in the first stage (see Figure 16). In this case, there is a path from a use of *v'* to a definition of *u'* in the transformed program and this is computed by step 4.4.

After the final interference graph is built, step 4.5 attempts to color it using existing heuristics in the literature, and finally, the steps 4.6 and 4.7 set up the transmission properly in the current and the next stages, as illustrated by Figure 12.

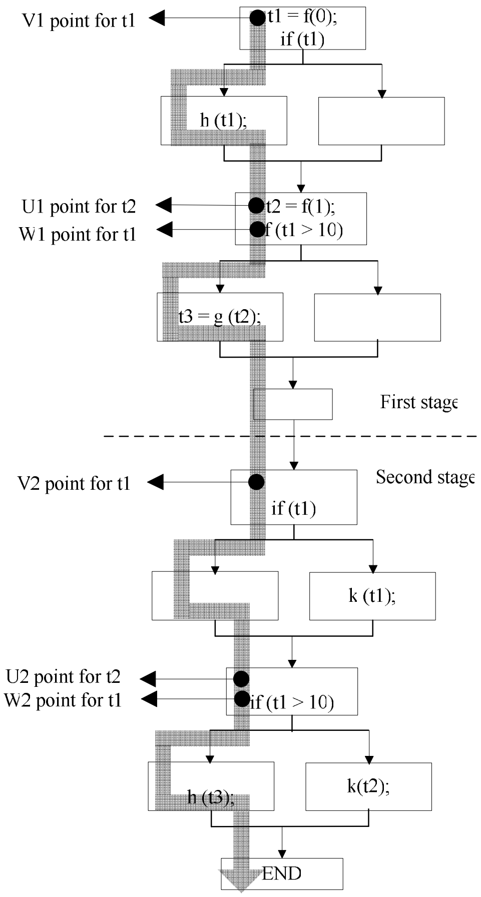


Figure 15. The first case for interference: V1→U1→W1→V2→U2→W2

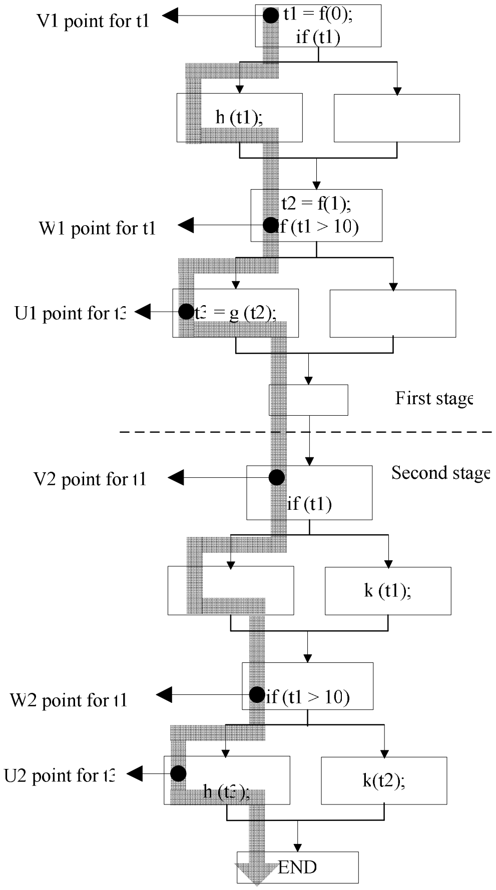


Figure16. The second case for interference: V1→W1→U1→V2→W2→U2

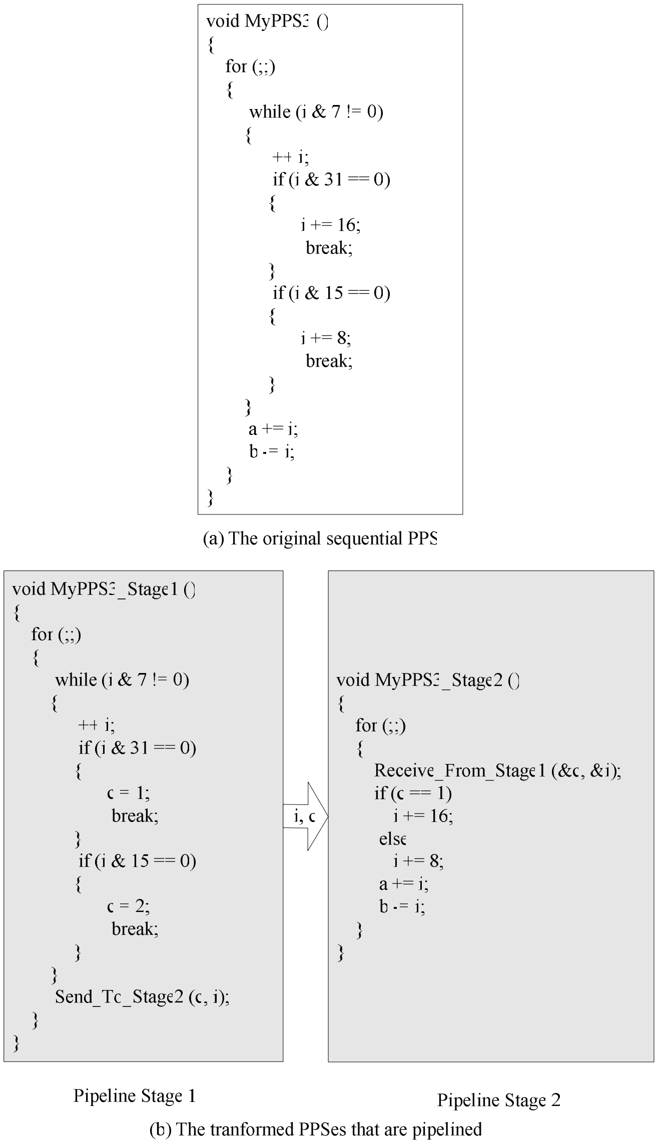
3.4.2 Reconstruction of the control flow

The reconstruction of the control flow for the pipeline stage is largely straightforward. One subtlety here is that as the control dependence is built from the summarized CFG, the conditional in the summarized CFG can be a loop that contains multiple exits. In that case, a different value needs be assigned to the control object in every successor block of that loop in *step 3.4*; furthermore, the reconstruction of the conditional in *step 3.3* should replace the loop by conditional branch (switch) to the appropriate successor block based on the associated control object. Such an example is shown in Figure 17.

4. Experimental results

The pipelining transformation can be applied to arbitrary network applications written using the auto-partitioning programming model. It has been implemented in the Intel auto-partitioning C compiler product and has been tested on several real-world applications in different network segments (e.g., broadband access, wireless, enterprise security, and core/metro network,).

In this section, we evaluate the effectiveness of the pipelining transformation using the industry standard Network Processor Forum (NPF) IPv4 forwarding benchmark [25] and IP forwarding benchmark (both for IPv4 and for IPv6) [26]. These two benchmarks selected for our experimental measurements because they are real and standard network applications and are widely used in the industry to test the system level performance of NPs.



```
void MyPPS3 ()
{
   for (;;)
   {
      while (i & 7 != 0)
      {
         ++ i;
         if (i & 31 == 0)
         {
            i += 16;
             break;
         }
         if (i & 15 == 0)
         {
            i += 8;
             break;
         }
      }
      a += i;
      b -= i;
   }
}
```

(a) The original sequential PPS

```
void MyPPS3_Stage1 ()
{
   for (;;)
   {
      while (i & 7 != 0)
      {
         ++ i;
         if (i & 31 == 0)
         {
            c = 1;
             break;
         }
         if (i & 15 == 0)
         {
            c = 2;
             break;
         }
      }
      Send_To_Stage2 (c, i);
   }
}
```

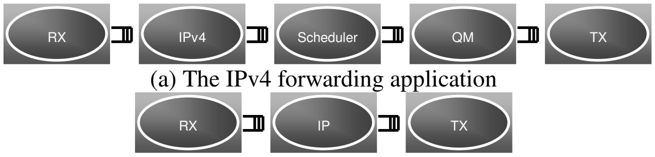
```
void MyPPS3_Stage2 ()
{
   for (;;)
   {
      Receive_From_Stage1 (&c, &i);
      if (c == 1)
         i += 16;
      else
         i += 8;
      a += i;
      b -= i;
   }
}
```

Pipeline Stage 1

Pipeline Stage 2

(b) The tranformed PPSes that are pipelined

Figure 17. Example for transfer of control flow after a loop



(a) The IPv4 forwarding application

(b) The IP forwarding application

Figure 18. The NPF benchmarks

The two applications are illustrated in Figure 18. The IPv4 forwarding application consists of five PPSes: the packet receipt (RX) PPS, the IPv4 PPS, the Scheduler PPS, the queue manager (QM) PPS, and the packet transmission (TX) PPS; and the IP forwarding application is made up of three PPSes: the RX PPS, the IP PPS and the TX PPS, with the IP PPS consisting of two code paths – one for the IPv4 traffic and the other for the IPv6 traffic. Each one has a complex control flow graph, with ~10K lines of codes, >600 basic blocks, ~100 routines, and >20 loops.

We evaluate the performance of each PPS in terms of the number of instructions required for processing a minimum sized packet (48 bytes for Packet Over SONET) for the IPv4 traffic and/or the IPv6 traffic, as this case places the most stringent performance requirement on the application.

The effectiveness of the pipelining transformation is evaluated by studying the speedup of the performance (i.e., comparing the performance of n-way pipelining a PPS with that of mapping it to a single PE), as well as the overhead of the live set transmissions, with different pipelining degrees. When measuring the performance of a particular PPS with pipelining degree *d*, the PPS is first *d-way* pipelined, and then the number of instructions required is determined by the longest pipeline stage.

In addition, the overhead of the live set transmissions is measured by the ratio, in the longest pipeline stage, of the number of instructions for live set transmission (receiving the live set from the previous stage and transmitting the live set to the next stage) to the number of instruction counts for packet processing.

Figures 19 and 20 show the speedup of the PPSes in the IPv4 forwarding and IP forwarding applications for different pipelining degrees. The speedup of the RX and TX PPSes, in both the IPv4 forwarding and IP forwarding applications, scales well up to pipelining degree 5, after which the speedup levels off. This is due to the fact that, as the pipelining degree increases, the reduction in the number of instructions in each pipeline stage is offset by the additional instructions required for live set transmission, as can be seen in Figures 21 and 22.

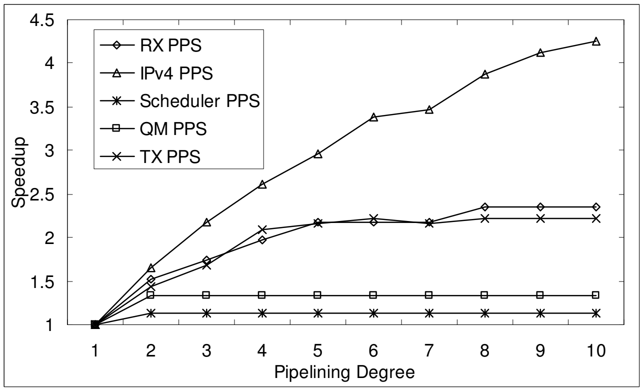


Figure 19. Speedup in the IPv4 forwarding benchmark

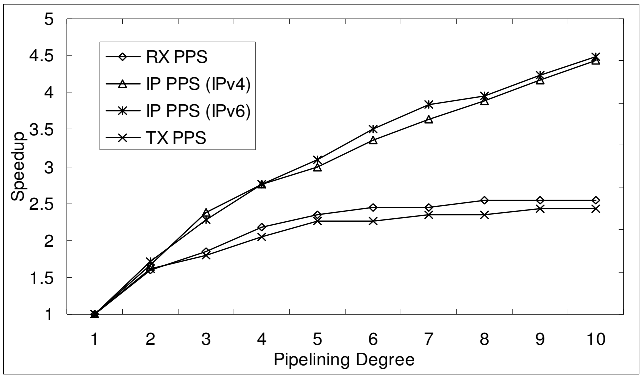


Figure 20. Speedup in the IP forwarding benchmark

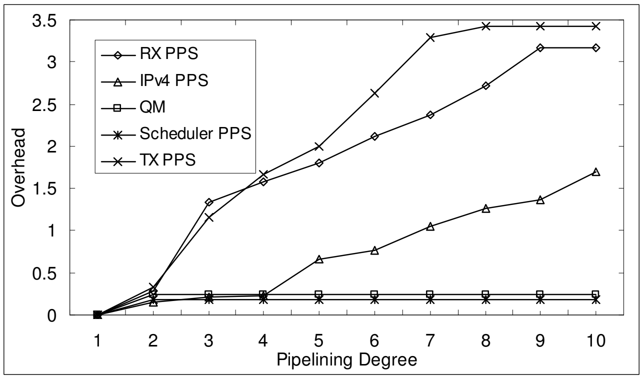


Figure 21. Overhead in the IPv4 forwarding benchmark

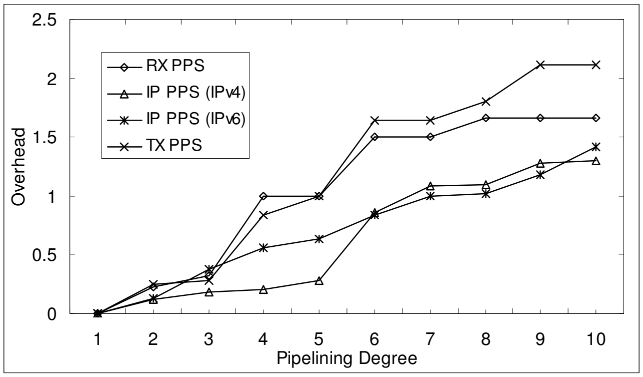


Figure 22. Overhead in the IPv4 forwarding benchmark

On the other hand, the speedup for the IPv4 PPS scales well for pipelining degree up to 10, as the number of instructions for packet processing in this PPS is much larger than that for live set transmission (as can be seen in Figure 21). The same is true for the IP PPS (for both the IPv4 traffic and the IPv6 traffic).

In contrast, the speedup for the QM and the Scheduler PPSes is almost the same for pipelining degrees 2 to 10. Since those two PPSes essentially update the shared flow state of the traffic, they have inherent PPS loop-carried dependence in the program. Consequently, they cannot be effectively pipelined (though they can be efficiently multi-threaded using one PE [7]).

Network applications usually have inherent data parallelism, i.e., they perform largely independent operations on successive packets, and hence they have little PPS-loop-carried dependence. They also have very stringent performance budgets (cycles per packet). For this reason the pipelining transformation is both useful and effective for improving the performance of network applications, as it distributes the performance budget over several pipeline stages.

5. Related work

Program partitioning is a heavily researched topic. Zhang et. al. [15] introduce a whole program partitioning method for tamper-resistant embedded devices, in which program partitions are generated under the principle of concealing the program control information to avoid security hazard. All program partitions are executed later in the same embedded devices.

A large volume of literature exists on mapping and scheduling parallel programs for multi-processor systems [9][10][13][18][19][20][21][22][23][24]. Choudhary et. al. [21] address the problem of optimal processor assignment to a pipeline of coarse grain tasks but assume no communication cost (or that the communication cost can be folded into computation cost). In [20], Subhlok and Vondran introduce a method to perform optimal mapping of *k* tasks onto *P* processors while taking communication cost into consideration. However, all the above work focus on how to partition existing task chains into multi-processors, while no work has been done on the problem of partitioning a whole program into chained tasks.

Loop distribution [16][17] performs a similar program transformation to that presented here: it partitions a loop into several loops. Typically, *scalar expansion* is used to communicate live quantities between the resulting loops; the expanded scalars (vectors) correspond to our *pipes*. Note that loop distribution cannot be applied to infinite loops, whereas the transformation we have outlined is applied *exclusively* to infinite loops (PPS loops). Loop distribution typically results in several loops that are run one after the other on a single processor; in our technique the resulting pipeline stages are intended to be run simultaneously on multiple processors, with pipes acting both as inter-processor communication and as synchronization.

Although in name the technique of *software pipelining* would seem to be closely related to the transformation presented here, both the aim and the effect of software pipelining are quite different from our context pipelining. Software pipeline rearranges the body of a loop so as to take advantage of inter-iteration parallelism to tolerate latencies in functional units and memory accesses, and to eliminate resource-based scheduling hazards (see [14] and [30] for some representative examples of software pipelining). It is *not* the goal of software pipelining to put multiple processors into use on a single loop, nor is it the goal of software pipelining to split what was originally a single loop into multiple loops, nor does software pipelining introduce any inter-loop communication or synchronization along the lines of our pipes. Software pipelining is motivated by the abundance of *instruction-level parallelism* that is made available by rearranging a loop body into pipeline stages taking into advantage inter-iteration independence. The resulting loop body is executed on a single processor, in a single instruction stream. In contrast, our pipeline stages are intended to execute asynchronously on multiple processors using communication and synchronization; this corresponds much more closely to loop distribution than to software pipelining.

There are some other proposals in the literature with similar goals; however, their approaches are very different. For instance, in StreamIt [27] the pipeline and parallelism constructs are explicit in the source code; on the other hand, the PPSes are unannotated sequential C program. The DEFACTO system [28] focuses on coarse-grain inter-loop pipeline, and the work by Du *et al.* [29] focuses on structured control flow and constructs (e.g., *foreach* loop); those are very different from our approach that works on the complex and arbitrary control flow inside the PPS loop.

The processing engines in the network processors can be also employed as a pool of homogenous processors operating on distinct packets. The auto-partitioning C compiler is also capable of replicating a single PPS, so that the same PPS runs on multiple threads and PEs, by inserting proper synchronization codes [7].

There are complicated tradeoffs in the resource management, in addition to the code size implications, between these two approaches. In brief, pipelining transformation permits the resources of a PE to be divided among the *stages* of the pipeline as opposed to dividing them among the *instances* of the computation applied to individual packets. The performance result may be radically different as a result.

6. Concluding remarks and future works

In this paper, we present a method that automatically partitions a program with data parallelism into several chained sub-tasks (pipeline stages) that can be mapped onto pipelined multiprocessor architectures. In our approach, the balance of the packet processing tasks is taken into account during program partitioning, and the data transmission between chained sub-tasks is minimized.

The original program is modeled using a flow network, and balanced minimum cuts are found on this flow network. Finally, each pipeline stage is realized with the minimum live set transmissions. The transformation is effective in improving the performance of packet processing applications by distributing the processing tasks over several pipeline stages while minimizing the overhead of live set transmission.

Although we illustrate our algorithms using packet processing applications as examples, the methods described in this paper can be applied to other data parallel programs such as digital signal processing, imaging processing and computer vision as well.

In our transformation, how the placement of the codes affects the overall balance of the packet processing tasks is modeled using a weight function. It is flexible and can model various factors (e.g., instruction count, instruction latency, hardware resources, or combinations thereof). In our implementation, it is used to distribute the instruction count; based on the encouraging results from this paper, we would like to extend it to explore the effect of distributing IO latency and hardware resource (e.g., CAM and local memory [2]) over pipeline stages.

7. Acknowledgements

The authors would like to thank Wen-Hann Wang for providing insightful suggestions on revising the paper. The authors would also like to thank the anonymous reviewers for their constructive comments that helped improve the final paper.

8. References

[1] *Challenges in Building Network Processor Based Solutions*, http://www.futsoft.com/pdf/NPwp.pdf

[2] *Intel IXP family of Network Processors*, www.intel.com/design/network/products/npfamily/index.htm

[3] *IBM PowerNP Network Processors* http://www-3.ibm.com/chips/techlib/techlib.nsf/products/IBM_PowerNP_NP4GS3

[4] *CPort Network Processor family,* http://www.windriver.com/cgi-bin/partnerships/directory/viewProd.cgi?id=1371

[5] *Agere's PayloadPlus Family of Network Processors*, http://www.agere.com/telecom/network_processors.html

[6] *AMCC's nP7xxx series of Network Processors*, http://www.mmcnetworks.com/solutions/

[7] *Introduction to the Auto-Partitioning Programming Model*, http://www.intel.com/design/network/papers/25411401.pdf

[8] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall International Series in Computer Science, 1985. ISBN 0-13-153271-5 (0-13-153289-8 PBK).

[9] *TejaNP*: A Software Platform for Network Processors*, http://www.teja.com

[10] Vin, H., Mudigonda, J., Jason, J., Johnson, E., Ju, R., Kunze, A. and Lian, R. *A Programming Environment for Packet-processing Systems: Design Considerations*, 3rd Workshop on Network Processors & Applications (Feb. 2004)

[11] Michael K. Chen, Xiao-Feng Li, Ruiqi Lian, Jason H. Lin, Lixia Liu, Tao Liu, and Roy Ju. *Shangri-la: Achieving high performance from compiled network applications while enabling ease of programming*, In *Proceedings of ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*

[12] Goldberg, A.V. and Tarjan, R.E. *A new approach to the maximum flow problem*. In *Proc. 18th ACM STOC* (1986), 136-146

[13] Yang, H. and Wong and D. F. *Efficient network flow based min-cut balanced partitioning*. In *Proc. IEEE Intl. Conf. Computer-Aided Design* (1994), 50-55

[14] Lam, M. *Software pipelining: an effective scheduling technique for VLIW machines*. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation* (June 1988), Volume 23 Issue 7

[15] Zhang, T., Pande, S. and Valverde, A., *Tamper-resistant whole program partitioning, In ACM SIGPLAN Notices* , In *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems* (June 2003), Volume 38 Issue 7

[16] Bacon, David F., Graham, Susan L. and Sharp, Oliver J., *Compiler transformations for high-performance computing*, in *ACM Computing Surveys (CSUR)*, Volume 26 Issue 4 (Dec. 1994)

[17] Kennedy, K. and McKinley, Kathryn S., *Loop distribution with arbitrary control flow*, In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing* (Nov. 1990)

[18] Han, Jia L., *Program partition and logic program analysis*, In *IEEE Transactions on Software Engineering*, Volume 21 , Issue 12 (Dec. 1995), 959 – 968

[19] Yang, T. and Gerasoulis, A., *PYRROS: static task scheduling and code generation for message passing multiprocessors*, In *Proceedings of the 6th international conference on Supercomputing*, p.428-437, July 19-24, 1992, Washington, D. C., United States

[20] Subhlok, J. and Vondran, G., *Optimal mapping of sequences of data parallel tasks*, ACM SIGPLAN Notices, v.30 n.8, p.134-143, Aug. 1995

[21] Choudhary, A. N., Narahari, B., Nicol, D. M., and Simha, R., *Optimal Processor Assignment for a Class of Pipelined*

Computations, In *IEEE Transactions on Parallel and Distributed Systems*, v.5 n.4, p.439-445, April 1994

[22] Subhlok, J., O'Hallaron, David R., Gross, T., Dinda, Peter A., and Webb, J, *Communication and memory requirements as the basis for mapping task and data parallel programs*, In *Proceedings of the 1994 conference on Supercomputing*, p.330-339, December 1994, Washington, D.C., United States

[23] Orlando, S. and Perego, R., *Scheduling Data-Parallel Computations on Heterogeneous and Time-Shared Environments*, In *Proceedings of European Conference on Parallel Processing*, Pages 356-366, 1998

[24] Gordon, M. I.., Thies, W. , Karczmarek, M., Lin, J., Meli, A. S., Lamb, A. A., Leger, C., Wong, J., Hoffmann, H., Maze, D. and S. Amarasinghe, *A Stream Compiler for Communication-Exposed Architectures*, in *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October, 2002.

[25] Network Processor Forum (NPF), *IPv4 Forwarding Benchmark Implementation Agreements (July 2002)*, http://www.npforum.org/benchmarking/licenseagm_IPv4.shtml

[26] Network Processor Forum (NPF), *IP Forwarding Benchmark Implementation Agreements (June 2003)*, http://www.npforum.org/benchmarking/licenseagm_ipforwarding.shtml

[27] William Thies, Michal Karczmarek, Michael Gordon, David Maze, Jeremy Wong, Henry Hoffmann, Matthew Brown, and Saman Amarasinghe. *StreamIt: A Compiler for Streaming Applications*, *MIT-LCS Technical Memo TM-622*, Cambridge, MA (December, 2001)

[28] Heidi Ziegler, Byoungro So, and Mary Hall Pedro Diniz. *Coarse-Grain Pipelining for Multiple FPGA Architectures*, In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April, 2002

[29] Wei Du, Renato Ferreira, and Gagan Agrawal, *Compiler Support for Exploiting Coarse-Grained Pipelined Parallelism*, In *Proceedings of the ACM/IEEE SC2003 Conference*, 2003

[30] Rau, B. R. *Iterative modulo scheduling: an algorithm for software pipelining loops*. In Proceedings of the 27th annual international symposium on Microarchitecture (1994), ACM Press, pp. 63--74.