Program Mapping onto Network Processors by Recursive Bipartitioning and Refining

Jia Yu, Jingnan Yao, Laxmi Bhuyan
University of California Riverside
Riverside, CA 92521
{jiayu, jyao, bhuyan}@cs.ucr.edu

Jun Yang
University of Pittsburgh
Pittsburgh, PA 15261
junyang@ece.pitt.edu

ABSTRACT

Mapping packet processing applications onto embedded network processors (NP) is a challenging task due to the unique constraints of NP systems and the characteristics of network application domains. A remarkable difference with general multiprocessor task scheduling is that NPs are often programmed into a hybrid parallel and pipeline topology.

In this paper, we introduce a multilevel balancing and refining algorithm for NP program mapping. We use a divide-and-conquer approach to recursively bipartition the task graph into disjoint subdomains. At each level of bipartition, the processing resources will be co-allocated so that an estimation of throughput can be derived. The bipartition continues until the code of the tasks can be fit into the instruction memory of processing elements. Then the algorithm iteratively refines the solution by migrating tasks from the bottleneck stage to other stages. The performance of our scheme is evaluated with a suite of NP benchmarks using SUIF/Machine SUIF compiler and Intel IXA Architecture Tool. The throughput improvement is significant: average throughput is increased by 20%, and the maximum is 108%.

Categories and Subject Descriptors: C.3: Special purpose and Application-based Systems

General Terms: Algorithms, Performance

Keywords: Network Processors, Program Mapping

1. INTRODUCTION

The growth of Internet has required not only an order-of-magnitude increase in the forwarding capacity of routing equipment, but also support for a wider spectrum of applications, highlighting the need for scalable router design and architecture. Network processors (NP), with programmability and short time-to-market, have emerged as a new alternate solution to next generation routers.

Although the idea of building programmable devices is appealing, the complex hardware and specialty of packet processing applications make NP programming a challenging task. NPs incorporate multi-core and/or multi-threading architecture with exposed multi-level memory hierarchy. The instruction memory on each processing element (PE) has limited space due to relatively high chip area of memory compared to processing logic. For instance, Intel IXP2800 contains 16 PEs, and each PE has an instruction memory holding up to 8K instructions [1]. The limited instruction memory space cannot accommodate the increasingly versatile and complex applications, such as multimedia transcoding, VoIP, and TCP offloading. Therefore the applications have to be partitioned into micro-tasks which can be stored into separate PEs. Then, the PEs executing different micro-tasks form a coarse level pipeline. Limited local data memory is not considered as the first-order constraint, because NPs usually store bulk packet data and routing tables in off-chip memory, and use multi-threading to hide memory latencies.

A distinct feature with an NP system is that multiple PEs can be configured to run the same task, even within one pipeline stage. Those PEs execute the same code to process different packets concurrently, leveraging the abundant *packet level parallelism (PLP)* in network applications. Thus an NP application can be mapped onto a pipeline topology with parallel sub-topology in some stages. Fig. 1 illustrates an example of such hybrid parallel and pipeline programming model. Stage 2 contains task 2 and 3. It is allocated with three PEs while stage 1 and 3 only has one PE respectively.

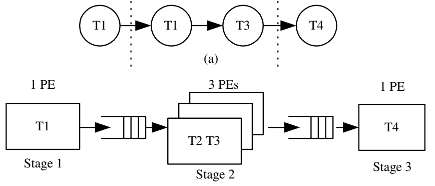

Figure 1: (a) Program partition (b) Processing resource mapping

The throughput of the pipelined program mapping is determined by the *maximum stage latency*, which includes both the processing and the inbound/outbound communication time. A good mapping scheme should minimize the *maximum stage latency* to achieve the highest throughput. Previous work [2, 4, 5] took the approach to duplicate PEs in the bottleneck stage to reduce its effective stage time. However, since PEs are not divisible, achieving true balanced stage time would require a large number of PEs, resulting inefficient utilization of PE resources. Also, previous work either neglect the communication cost [2], or do not keep the number of stages low so that the aggregated communication cost is high [4, 5].

In this paper, we propose to improve the throughput through

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2007, June 4–8, 2007, San Diego, California, USA.
Copyright 2007 ACM ACM 978-1-59593-627-1/07/0006 ...$5.00.

both reducing the number of pipeline stages with minimum aggregated communication cost, and reducing the maximum stage time. We use a recursive bipartition algorithm to restrain the stage numbers while keeping the aggregated communication cost minimum. To reduce the maximum stage time, we propose to use stage refinement to migrate tasks from bottleneck stages to non-bottleneck stages in the framework of our recursive bipartitioning. Such an approach is more effective and precise to reduce the bottleneck stage time. Also, it does not require excessive PE resources, and can achieve better throughput with the same number of PEs. Our methodology is particularly appealing when a large number of applications are mapped onto limited PEs, or when the PEs for main packet processing are reduced due to dedicated PEs on routine tasks, such as packet receiving/transmitting and queue management. Experimental results show that our recursive bipartitioning and refining algorithm improves throughput by about 20% on average and 108% at maximum, compared with previous algorithms.

The remainder of paper is organized as follows. Section 2 describes the problem formulation and existing mapping algorithms. Section 3 introduces the recursive bipartitioning and refining algorithm with its extension for multi-application mapping. Section 4 presents the simulation results. Finally, Section 5 concludes this paper and discusses the future work.

2. PROBLEM FORMULATION AND EXISTING ALGORITHMS

2.1 Problem Formulation

A program is characterized by a program dependence graph (PDG). The PDG can be constructed from the control flow graph (CFG) following Ferrante's algorithm [6]. An example of a CFG and its corresponding PDG is shown in Fig. 2. This PDG contains six basic blocks (in circles and diamonds) representing $B1$ to $B6$ in the CFG. They are taken as tasks. The "region" vertices $\{Entry, R2, R3, R4\}$ (in pentagons) summarize the set of control conditions for a task. For example, $B5$ is executed under the control condition $\{4T\}$. Hence a region vertex $R4$ is inserted to represent $\{4T\}$, and $B5$ is a child of $R4$. The dashed arrows with numbers represent the communication costs between tasks. The solid arrows with labels T or F, represent $true$ or $false$ control dependencies. This example does not have an $else$ statement, so we only observe control dependencies with T condition in the PDG.

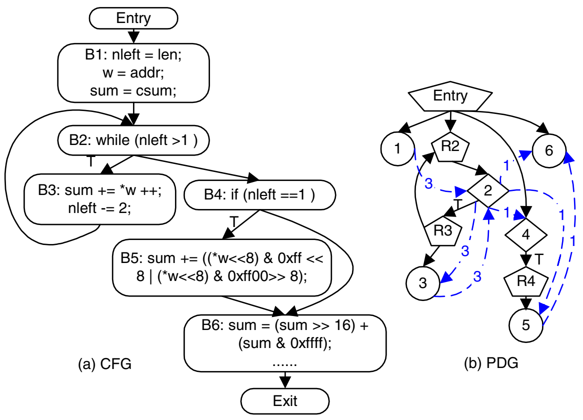

Figure 2: The CFG and PDG of a snippet of IPv4 forwarding application

Let N be the number of PEs in an NP. The N PEs are arranged into k pipeline stages. We denote ST_i as the actual stage time of stage S_i. ST_i reflects the latency of an individual packet in S_i, which includes task execution time, inbound and outbound communication latency:

$$ST_i = \sum_{j \in S_i} E_j + \sum_{i \notin S_i, j \in S_i} C_{ij} + \sum_{i \notin S_i, j \in S_i} C_{ji} \quad (1)$$

The first term is the total execution time of tasks in S_i (E_j is the execution time of task j). The second and third terms are the inbound and outbound communication cost respectively (C_{ij} is the weight of the dashed line from task i to j in the PDG). Here we assume an ideal case with multi-threading, where the PEs are 100% utilized. We will augment the stage time model with multi-threading effect in future work.

We denote ST_i^e as the effective stage time, which is defined from the stage throughput point of view. That is, every ST_i^e time, a packet can be released from stage S_i. ST_i^e can be determined by the actual stage time and the number of PEs assigned to this stage, P_{S_i}:

$$ST_i^e = \frac{ST_i}{P_{S_i}} \quad (2)$$

The throughput of a pipelined mapping is determined by the slowest stage, i.e.,

$$Throughput = \frac{1}{Max_{i=0}^{k-1}\{ST_i^e\}} \quad (3)$$

Thus the goal of NP mapping is to minimize the *max effective stage time*. During this procedure, some constraints should be considered [4]: (i) The code size of tasks in each pipeline stage should be less than the instruction memory size. (ii) Minimize the communication cost across stages, and no backward dependencies should exist across pipeline stages. Otherwise, the earlier stage will be required to stall in the middle of execution until the data comes in from a later stage. Backward-flow synchronization is very expensive, so it is usually avoided in mapping. For loops, where backward dependencies usually happen, we consolidate those tasks into big tasks and avoid partitioning them across stages.

2.2 Existing Algorithms

In general, optimal partitioning and mapping is an NP-hard problem [2]. Stream programming offers heuristic partitioning algorithms (i.e. greedy heuristic) to attain high performance [3]. However, these algorithms cannot be applied to NP domain directly, because they rely on language support that exposes coarse-grained parallelism and the constraint of limited instruction memory is not considered. Therefore, we do not compare NP partitioning algorithms with stream programming in this paper.

For NP program mapping, Wolf et al. [7] describe an iterative randomization algorithm to map the tasks. This algorithm is not scalable with increasing number of tasks in the graph, because the search space is too big, and significant time is spent filtering out invalid mappings that violate dependency constraints.

Yao et al. [2] proposed to greedily pack the tasks in sequential order until the code size in a stage exceeds instruction memory size. After code packing, the algorithm allocates PEs to stages in proportion to the actual stage time. We call this algorithm a *code greedy* algorithm. This algorithm does not consider the tradeoff of code space and communication cost in the code packing. Thus it may generate a mapping which has high communication cost.

Intel IXP auto-partitioning C compiler [4] used a min-cut algorithm $k-1$ times to divide the program into k sequential parts of roughly the same size. The k parts are then mapped to PEs to form a k-stage pipeline. The min-cut algorithm automatically minimizes communication costs across edges.

However, the algorithm assumes that the number of stages k is given, and the programs are equally divided into k parts, thus is not directly applicable to hybrid parallel and pipeline model, which allows heterogeneous actual stage times with a balanced *effective* stage time.

Shangri-la [5] uses a throughput-driven heuristic algorithm to merge tasks into stages from bottom up. Their heuristic is "if the throughput at step i increases by zeroing the highest edge cost then zero this edge." Shangri-la algorithm is suitable for coarse granularity graphs which have very non-homogeneous edge costs. If the edge costs are same or similar, the algorithm will lose the hint of which edge to merge and thus generate sub-optimal pipeline mapping.

Our algorithm uses the concept of partitioning with control on the pipeline stage number and the aggregated communication cost. We do not require that the stage number is known apriori. Moreover, our algorithm automatically generates hybrid parallel and pipeline topology after task graph partitioning and PE resource mapping.

3. PROPOSED RECURSIVE BIPARTITIONING AND REFINING

In this section, we describe the recursive bipartitioning of the task-to-processor mapping, and a local refinement mechanism of migrating tasks from bottleneck stages to other stages.

3.1 Resource Balanced Bipartitioning

To minimize the aggregated communication cost of the entire pipelining, we adopt the divide-and-conquer approach. That is, we bipartition the program recursively into fewest number of stages with minimum communication cost during each partition. The recursive bipartioning serves for the purpose of finding the optimal number of stages. And the minimum aggregated communication cost is achieved by applying the min-cut in each step of partition. The bipartition algorithm has been studied in multiprocessor task allocation for achieving balanced parallel task execution and minimal total execution time [10]. We extend this algorithm to our problem to achieve balanced pipelining and maximum throughput. The procedure is given in Fig. 3.

```
Resource Balanced Bipartitioning
Input: Task graph $G(T,E)$ and number of PEs $N$.
Compute the cut_ratio for bipartitioning.
Call the r-Balanced Min-Cut procedure over task graph $G(T,E)$ and get subgraphs $G_1(T_1,E_1)$, $G_2(T_2,E_2)$.
Compute actual stage time of two partitions ($G_1$, $G_2$) using Formula (1) and allocate $N_1$, $N_2$ PEs to them.
Insert $G_1$, $G_2$ into the partition tree T.

/* Recursively partition */
if ($G_1$'s code size > IM size) then
    Call Resource Balanced Bipartitioning with $G_1$ and $N_1$.
if ($G_2$'s code size > IM size )then
    Call Resource Balanced Bipartitioning with $G_2$ and $N_2$.
```

Figure 3: The resource balanced bipartitioning procedure

Our bipartition algorithm is based on the *r-Balanced Min-Cut* which cuts off vertices whose total weights are of fraction r of the total graph weight [11]. This algorithm is adopted from the iterative balanced push-relabel algorithm which is widely used to find Max-Flow-Min-Cut [12]. r is the *cut_ratio* between two partitions with the default value of 0.5. If the estimated number of stages is odd, r should be adjusted accordingly. For example, r can be adjusted to $2:3=0.4$ (or 0.6) for a *5-stage* pipeline in the bipartition procedure.

The initial value of r should be a tight estimation. That is, if the program needs at least p partitions, then r should be set according to p. Otherwise, the algorithm would find more stages than necessary. We use the minimum stage number to derive the initial estimation of r. The minimum stage number is the total program code size divided by the instruction memory size per PE, as shown in (4).

$$MIN\_stage\_number = \lceil \frac{\sum_{i \in T} M_i}{IM\ \ Size} \rceil \qquad (4)$$

where M_i is the code size of task i. Note that the final optimal number of stages is not necessarily the minimum stage number. Other factors including task granularity, communication variance might bias the cut and increase the number of stages. The bipartition procedure takes into account these factors and adjusts the stage number accordingly.

After a min-cut is performed, we compute the actual stage time of the two subgraphs using (1). Then we partition the N PEs into N_1, N_2 and allocate them to the two subgraphs. We first allocate enough PEs to accommodate the static code. However, this allocation does not necessarily balance the execution time between the two subgraphs. Therefore, we allocate remaining PEs in proportion to the actual stage time to form parallel execution. This also helps minimize the effective stage time because the more time consuming a stage is, the more PEs should be allocated to it to balance the pipeline workload. Equation (5) and (6) show the lower bound and upper bound of number of PEs that should be allocated to partition 1 and 2 respectively.

$$N_1 \in [\lceil \frac{\sum_{i \in T_1} M_i}{IM\ \ Size} \rceil,\ \ \lceil N * \frac{ST_{part1}}{ST_{part1}+ST_{part2}} \rceil] \qquad (5)$$

$$N_2 \in [\lceil \frac{\sum_{j \in T_2} M_j}{IM\ \ Size} \rceil,\ \ \lceil N * \frac{ST_{part2}}{ST_{part1}+ST_{part2}} \rceil] \qquad (6)$$

During the recursive bipartitioning process, we maintain a partition tree, which will be used in the local refinement for pipeline workload balancing. Balanced trees [16], such as AVL tree, splay tree have similar property, but are used to balance heights of the two child subtrees. In our approach, we balance the throughput in the two child subtrees.

The bipartition procedure continues until each partition's code size is less than the size of instruction memory. After we derive all the leaves in the partition tree, we get an initial mapping in a hybrid parallel and pipeline topology.

3.2 Refinement

The initial mapping we derive from recursive bipartitioning is often suboptimal in workload balancing due to combinatorial constraints. For example, suppose the stage time ratio between two stages is 2:3, but the PE number ratio is 1:2 because only integer number of PEs can be allocated to the two stages. As a result, their *effective* stage time is 2:1.5 while a perfect ratio is 1:1. Previous work [2, 4, 5] rely solely on allocating more PEs to heavy loaded stages to achieve balanced pipeline. Due to the indivisibility of the PEs, such an approach is insufficient to attain an optimal balanced pipeline.

We adopt a refinement algorithm to improve our initial mapping result. Refinement is widely used in adjusting tasks among vertices in a general network. Well known algorithms such as the Kernighan-Lin's algorithm [13] and its subsequent extensions can be applied to our hybrid parallel and pipeline mapping with some modifications. Although we generally refer to the refinement algorithm as a local search, the paradigm does not preclude the use of more complex techniques, such as simulated annealing, genetic algorithms, etc.

Our essential idea is to migrate tasks from bottleneck stage to non-bottleneck stage. The simplest method is to perform *local adjustment*, as illustrated in Fig. 4 (a). Here we move tasks from the bottleneck stage only to its neighbors. This works well when the neighbors have sufficient instruction memory capacity and computation power. Otherwise it may fail. For example, the processing pressure in S5 cannot be mitigated if S4 and S6 cannot accommodate additional tasks due to full instruction memory or high workload. To make the migration global, we propose a *hierarchical adjustment* that combines the local adjustment with our partition tree. It performs local adjustment within a subtree, and gradually increases the tree level until a migration is successful. A successful migration is one that increases the overall throughput. For example, in Fig. 4 (b), S5 first tries to move tasks to S6. If unsuccessful, S5 passes the processing pressure to the parent node, which checks if the left subtree (S3, S4) can take the additional tasks without deteriorating the overall throughput. If S3 and S4 cannot accommodate tasks from S5, the algorithm goes up another level to probe S1, S2.

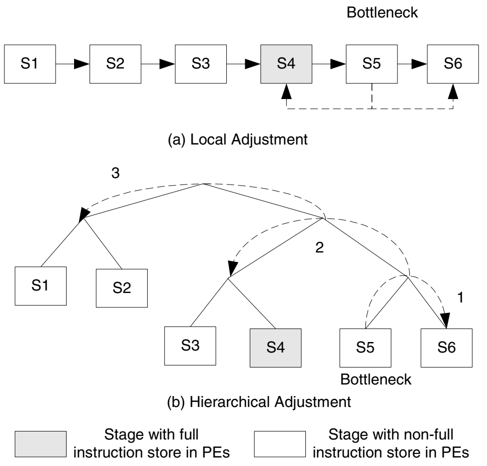
Bottleneck
S1
S2
S3
S4
S5
S6
(a) Local Adjustment
3
2
1
S1
S2
S3
S4
S5
S6
Bottleneck
(b) Hierarchical Adjustment
Stage with full instruction store in PEs
Stage with non-full instruction store in PEs

Figure 4: Two methods to perform refinement

During the hierarchical adjustment, tasks are migrated within a subtree subject to the precedence constraint of the program. Only the tasks at the stage boundary can be moved into neighbors. That is, tasks near the front (back) boundary can only be moved to the upstream (downstream) neighbor. For example in Fig. 4, when S5 cannot move the back boundary tasks to S6, it will try the sibling subtree (S3, S4). This adjustment may require S3 taking some front border tasks from S4 and S4 taking some front border tasks from S5. The boundary tasks are defined as:

$$T_i \in Front_{S_i} \;\; iff \;\; \forall T_i \in S_i, \;\; T_j \in S_i, \;\; (T_j, T_i) \notin E \tag{7}$$

$$T_i \in Back_{S_i} \;\; iff \;\; \forall T_i \in S_i \;\; T_j \in S_i, \;\; (T_i, T_j) \notin E \tag{8}$$

where $Front_{S_i}$ ($Back_{S_i}$) denotes the front (back) border tasks for stage i. For each task T_i considered for migration, we compute the throughput gain were T_i to migrate to some other stage. To verify if the sibling subtree can accommodate T_i, the algorithm calls the Bipartition procedure to repartition the graph after task migration. It migrates the task with the highest gain first. The refinement terminates when (i) the throughput of two subtrees are balanced within a tolerance or (ii) further task migration cannot help to improve throughput. The pseudocode of recursive local refinement is sketched in Fig. 5.

```
Local Refinement
Input: Partition tree $T$ and root node $R$.

/* Recursively refine, refine child nodes first */
if ($R$'s left subtree contains bottleneck stage) then
    Call Local Refinement with $R$ and $R$'s left child.
if ($R$'s right subtree contains bottleneck stage) then
    Call Local Refinement with $R$ and $R$'s right child.
do
    Compute the boundary tasks of bottleneck tree (left or
    right subtree).
    for (all tasks on the boundary)
        compute the throughput gain if shifting this task to
        sibling subtree.
    Migrate the task that maximizes the throughput gain.
    Update the partition tree $T$.
while ($|Throughput_{left} - Throughput_{right}| > \epsilon$ and $\exists$ task
on the boundary whose throughput gain $\geq 0$)
```

Figure 5: The local refinement procedure

3.3 Time Complexity

To bipartition an NP program into k stages, at most $k-1$ times of r-Balanced Min-cut will be executed. Based on Goldberg and Tarjan's work [12], r-Balanced Min-cut has $O(|T||E|)$ time complexity, where T, E are tasks and edges. Therefore, the complexity for our bipartition procedure is $O(k|T||E|)$. Considering that k is a small integer which is less than the number of PEs, we safely round the complexity to $O(|T||E|)$.

In the refinement procedure, the total number of task migration is $O(|T|)$, because we restrict task migration in one-way direction and the adjustments are only performed on the boundaries. There is little chance that the same task will be migrated twice, due to graphs' precedence constraints. In each task migration, the complexity of computing the boundary tasks of a partition is $O(|E|)$. A recursive bipartition might be called to calculate the throughput gain which has $O(|T||E|)$ complexity. Thus the total complexity of refinement is $O(|T| * (|E| + |T||E|))$.

Summing up the complexity of bipartition and refinement, we show that the total complexity is $O(|T|^2|E|)$. This time complexity should be acceptable, considering that program mapping is performed offline by compilers.

3.4 Extension to Map Multiple Applications

Our partitioning algorithm can be extended to map multiple applications. Instead of partitioning the applications individually, we merge the program dependency graphs of different applications into one unified graph. We add a virtual source vertex and a virtual sink node, which connect to the original source and sink nodes respectively. The weight of edges going out of the virtual source and coming into sink are set to 0. By doing so, the bipartition algorithm tends to cut these zero-cost edges for minimum cut cost. As a result, the tasks belonging to different applications will be partitioned into different stages, which can be executed concurrently. The application level parallelism can be maintained in the mapping results.

4. EXPERIMENT AND EVALUATION

In this section, we present the experimental results of our algorithm and highlight certain intrinsic advantages. We will first present the experimental framework, and then compare the throughput of our algorithm with two previous works and furnish sensitivity results of our algorithm.

4.1 Experimental Framework

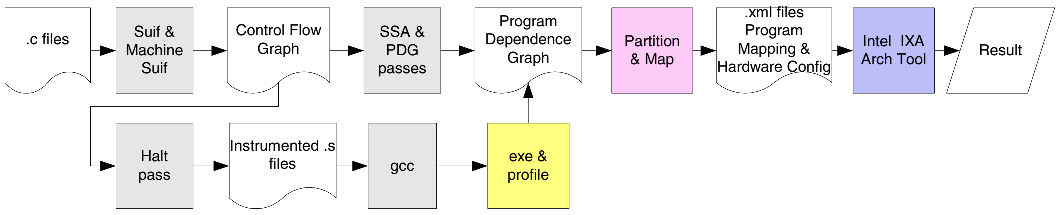

Figure 6: Experimental Framework

Table 1: Packet Processing Applications

Application	Functionality	Code Size(insn)
ipv4fwdr	Validation & trie-based lookup	1548
nat	Replacing address and port	1645
ipsec	AES encryption	3833
flow	Hashing based on 5-tuple	4215
portscan	Detecting suspicious activity	4760
md5	Integrity verification	4934

Fig. 6 shows our experimental framework which combines the SUIF/Machine SUIF [8][9] compilers and the Intel IXA Architecture Tool [1]. The original C programs are first converted to SUIF control flow graphs (CFGs). For the ease of data dependency analysis, we inlined all the major functions, which is a common approach in NP program construction [1]. Next we translate the intermediate representations (IRs) into the static single-assignment form, which facilitates the data flow analysis. We then wrote a new pass *pdg* to extract program dependence graphs (PDGs) based on control flow and data flow information. To profile the tasks' execution frequency, we use the Machine SUIF HALT library to instrument the program and then convert the IRs to assembly programs. Then the instrumented version of programs are compiled by *gcc*. The binaries are executed with continuous traffic traces to exploit the execution frequency of the tasks. Once the PDG with profiled information is acquired, different program mapping algorithms are applied to obtain the mapping results.

We evaluate the performance in Intel IXA Architecture Tool (AT) [1]. AT enables a full dynamic simulation of the application at the thread level. We use Intel IXP2800 [1]'s hardware configuration, which features 16 PEs running at 1.4GHz with 32MB SRAM and 512MB DRAM memory. Tasks are described by *I/O references* (references to Scratch ring, SRAM, DRAM memory units), *code blocks* (the number of PE instruction cycles taken by tasks), *signals* and *wait* (allows a wait on 0 or more signals). A group of tasks can be aggregated into a pipeline stage and duplicated according to the mapping results. The communication between pipeline stages is modeled as *Scratch Ring* and *SRAM* operations. The packet handler, control dependency information, contents of live registers will be considered in the communication cost between pipeline stages.

Six NP applications are ported from NetBench [15] and PacketBench [14]. Their code sizes are listed in Table 1 in ascending order. These code sizes will determine the pipeline depth. In order to measure the *worst-case* throughput, we feed unlimited continuous traffic with minimum packet size (i.e. 64B) to the simulations. The routing table used for the IPv4 forwarding is MAE-WEST [14].

4.2 Performance Results

We compare the throughput of our algorithm with two previous works: the *Code Greedy* [2] and the Intel IXP auto-partitioning C compiler [4]. We do not compare with the Shangri-la algorithm [5] because it assumes coarse granularity task graphs and may not produce valid mappings for PDGs. We vary the available PEs from 4 to 16 and instruction memory from 1K instructions to 5K instructions. These configurations closely match the specification of state-of-the-art NP products. The throughput is measured in *million packets per second (mpps)*.

Fig. 7 shows the throughput of three approaches with 8 PEs and instruction memory size of 1K instructions. We have six benchmarks and five combination benchmarks that can be accommodated by 8 PEs. The combination benchmarks represent situations where packets are processed by more than one application at the same time. The benchmarks are arranged in an increasing order of code size, in Fig. 7.

We observe that our bipartitioning with refinement performs better than code greedy [2] and balanced cut [4] algorithms. For 9 out of 11 benchmarks, our algorithm demonstrates higher throughput. The throughput improvement ranges from 6% to 108% with an average of 22%. For *flow*, our algorithm achieves the highest improvement of 108%. This is because *flow* has comparatively larger code size (as shown in Table 1), which requires at least 5 pipeline stages. Given 8 PEs, only 3 PEs are available for duplication purpose. Pure code duplication is no longer effective in balancing the workload in this case, thus balancing stage time in the hybrid parallel and pipeline model becomes critical.

For benchmarks *ipv4fwdr* and *nat*, we observe comparatively lower throughput improvement. This is because these two benchmarks have smaller code sizes, so the pipeline is not very deep. All the three approaches aggressively duplicate the pipeline stages and produce similar throughput. *Ipsec, portscan* and *md5* do not show much improvement even if their pipelines are deep, because they have big loops which are not divisible due to backward dependencies. Under such circumstances, no opportunity is left for refining the mappings.

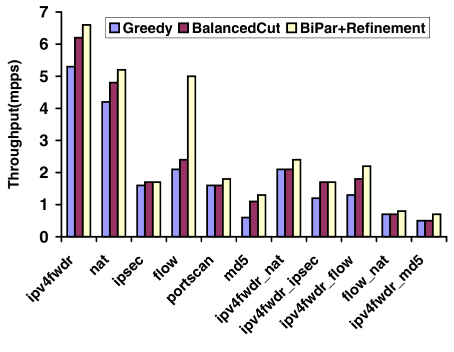

Figure 7: Throughput performance in three approaches, PE number=8, IM size=1K insns

Fig. 8 presents the throughput result with 16 PEs, 1K instruction memory. With more PE resource, the duplication possibility is significantly increased. Moreover, additional

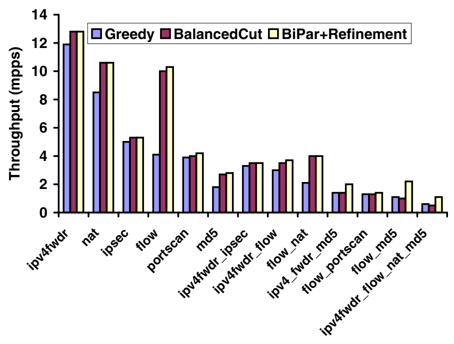

Figure 8: Throughput performance in three approaches, PE number=16, IM size=1K insns

benchmarks can be accommodated into a 16 PE NP. For the 13 benchmarks measured, we observe 19.3% throughput improvement on average. For single benchmarks (*ipv4fwdr* etc.), we observe little throughput improvement. This is because the three mapping algorithms have plenty of PEs for duplication, which are enough for balancing the pipeline's effective stage time. For combined benchmarks where resource constraint exists, we still observe significant throughput gain, i.e. 100% for *flow_md5*. These results demonstrate that our algorithm can effectively improve the NP throughput under resource constrained cases.

We vary the instruction memory size for *flow_md5* benchmark in Fig. 9, and illustrate the throughput increase with 16 PEs. Our refinement is very effective in balancing the pipeline workload when IM size equals 1K and 2K instructions. The throughput is increased from 1.4 mpps to 2 mpps with 1K instruction memory. As we increase the IM size to $5K$ instructions, the throughput improvement starts to level off. This is because the pipeline depth is significantly reduced with larger IM size. In the extreme case, the entire program can fit into one PE with rest PEs duplicated as parallel engines without pipelining.

Fig. 10 shows the packet throughput as a function of the number of PEs. Clearly the performance improves as more PEs are employed. As more PEs are recruited, the two previous algorithms also have enough PEs for code duplication. Hence the effect of refinement is less significant. However, our algorithm does show a consistent better throughput than the other two algorithms.

From Fig. 9 and 10, we observe that our recursive bipartitioning and refining algorithm is especially effective under resource constrained conditions. As we move to more complex applications on the Internet, code size of applications will increase and resources will be more constrained. The proposed algorithm will be more and more effective.

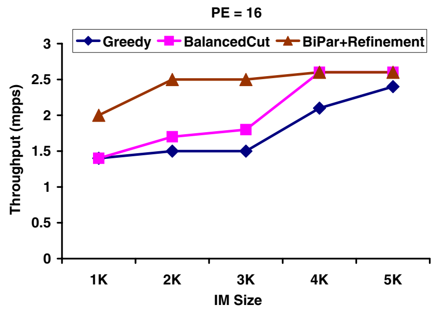

Figure 9: Throughput in three approaches with various sizes of IM, PE number=16, *flow_md5* benchmark

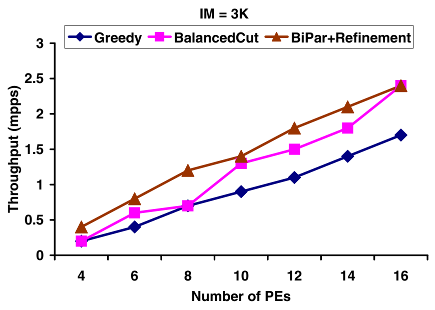

Figure 10: Throughput in three approaches with various number of PEs, IM size=4K insns, *flow_md5* benchmark

5. CONCLUSION

This paper addresses the challenges of task partitioning and mapping onto hybrid parallel and pipeline topology in NP systems. We propose recursive bipartitioning and refining to find a load balanced pipeline. The algorithm recursively migrates tasks from bottleneck stage to non-bottleneck stage to improve the throughput. Simulation results have shown that our algorithm can improve throughput by 20% on average with the state-of-the-art NP configurations. In our future work, we plan to exploit other factors that affect program mapping (i.e. data memory and heterogenous threads).

6. REFERENCES

[1] Intel IXP2XXX Product Line of Network Processors, Intel Corporation.
[2] J. Yao, Y. Luo, L. Bhuyan and R. Iyer "Optimal Network Processor Topologies for Efficient Packet Processing," *IEEE Globecom*, 2005.
[3] M.I. Gordon, W. Thies, and S. Amarasinghe "Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs," *ASPLOS*, 2006
[4] J. Dai, B. Huang, L. Li and L. Harrison, "Automatically Partitioning Packet Processing Applications for Pipelined Architectures," *PLDI '05,* pp. 237-248, 2005.
[5] M.K. Chen, X.F. Li, R. Lian, J.H. Lin, L. Liu, T. Liu and R. Ju "Shangri-La: Achieving High Performance from Compiled Network Applications while Enabling Ease of Programming," *PLDI '05,* pp. 224-236, 2005.
[6] Steven S. Muchnick, "Advanced compiler design and implementation," Morgan Kaufmann Publishers Inc., 1997
[7] N. Weng and T. Wolf "Pipelining vs. Multiprocessors – Choosing the Right Network Processor System Topology," *ANCHOR in conjunction with ISCA 2004.*
[8] SUIF Compiler System, Stanford University.
[9] Machine-SUIF, Harvard University.
[10] F. Ercal, J. Ramanujam and P. Sadayappan, "Task Allocation onto a Hypercube by Recursive Mincut Bipartitioning", *in Journal of Parallel and Distributed Computing* pp.35-44, Vol. 10, No. 1, 1990.
[11] H.H. Yang and D.F. Wong, "Efficient Network Flow Based Min-cut Balanced Partitioning" *in Proc. of the 1994 IEEE/ACM international conference on Computer-aided design* pp.50-55, 1994.
[12] A.V. Goldberg and R.E. Tarjan, "A New Approach to the Maximum Flow Problem" *in Proc. 18th ACM STO,* pp.136-146, 1986.
[13] B.W. Kernighan and S. Lin, "An efficient Heuristic Procedure for Partitioning Graphs," Bell Syst. Tech. J., pp. 291-308, Vol. 49, No. 2, 1970.
[14] R. Ramaswamy and T. Wolf. "PacketBench: A Tool for Workload Characterization of Network Processing," *WWC-6,* pp.42-50, 2003.
[15] G. Memik, W.H. Mangione-Smith and W.D. Hu "NetBench: A Benchmarking Suite for Network Processor," *ICCAD*, pp.39-, 2001.
[16] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, "Introduction to Algorithms," *MIT Press*, 2001.