Replication-Based Partial Dynamic Scheduling on Heterogeneous Network Processors

Zhiyong Yu[1], Zhiyi Yang[1], Fan Zhang[1], Zhiwen Yu[2], and Tuanqing Zhang[1]

[1] School of Computer Science, Northwestern Polytechnical University, P.R. China
yuzhiyong@mail.nwpu.edu.cn
[2] Academic Center for Computing and Media Studies, Kyoto University, Japan
yu@ccm.media.kyoto-u.ac.jp

Abstract. It is a great challenge to map network processing tasks to processing resources of advanced network processors, which are heterogeneous and multi-threading multiprocessor System-on-Chip. This paper proposes a novel scheduling algorithm, called Replication-based Partial Dynamic Scheduling (RPDS). It aims to improve the NP performance by combining the strategies of partial dynamic mapping and task replication with a 2-phase scheduling. RPDS differs from existing solutions in several aspects, e.g., the processing elements are heterogeneous, fully-connected, and multi-threading, the application is decomposed into directed acyclic graph tasks with continuous data-packets, and scheduling is conducted at both of initialization and run-time. Experimental results showed our algorithm could increase the largest average throughput by about 30% than those without dynamic phase replication.

Keywords: scheduling, network processors, task replication, partial dynamic scheduling, directed acyclic graph.

1 Introduction

The Internet has evolved from a simple store-and-forward network to a complex communication infrastructure. In order to meet demands on security, flexibility and performance of increasingly complex network services, network traffic not only needs to be forwarded, but processed on network devices such as routers. The programmable on-Chip Multi-Processors (CMP) called network processors (NPs) hence appeared. One of the difficulties of application development on such kind of hardware platform is to handle processing resources scheduling. And it also has some other restricts and requirements such as strong real-time, high throughput, low power, small instruction space, changing traffic load, etc., which make it a great challenge to solve this problem.

Scheduling of processing resources is basically to decide which task should be processed on given processing resource at a given time, to achieve the optimal goal. Within NPs, this problem is concretely the mapping from tasks to processing elements (PEs). The tasks are relatively independent code blocks which are decomposed from network applications by using two main methods, i.e. pipelining and directed acyclic graph (DAG).

M. Xu et al. (Eds.): APPT 2007, LNCS 4847, pp. 4–13, 2007.

The optimizing problem of mapping tasks to PEs is NP-complete [1]. So the practical goal is to get the approximate optimal result. Manual mapping is ineffective and fallible when the system architecture and application are complex [2], and previous research on automatic mapping did not consider the characteristics of advanced NPs.

According to weakest-link principle, the performance of the whole system relies on a few bottlenecks. So we can improve the system performance by abating them. When a task is identified to be a bottleneck, there are usually two solving methods, one is deepening the pipeline, which can't be changed after software compilation, the other is duplicating the task executable code to let it occupy more processing resources at the time of execution. If the bottleneck of a system is changing, the method of task replication can efficiently track the changes and abate the bottleneck.

Therefore, to map network processing tasks to processing resources of advanced complex network processors, this paper proposes a novel scheduling algorithm, called Replication-based Partial Dynamic Scheduling (RPDS). It combines the strategies of partial dynamic mapping and task replication together in NP scheduling that aims to improve the network processing performance in terms of throughput and delay.

The rest of this paper is organized as follows. In Sect. 2, we describe related work in NP scheduling and highlight the distinctive aspects of our approach. In Sect. 3, the details of problem formalization, processing models, and algorithm procedure are proposed. Section 4 presents the evaluation method, simulation tool and experimental results. Finally, Sect. 5 concludes the paper.

2 Related Work

Previous research of mapping tasks to PEs on NPs mainly utilized linear programming and heuristic algorithms, e.g., list scheduling, randomized mapping, and genetic algorithms. In linear programming method, the mapping problem is transformed to a linear programming problem to handled through greedy heuristic [3] or randomized rounding [4]. The list scheduling sorts all tasks according to their priorities and chooses a PE for each task based on a particular rule. Ramaswamy et al. [5] use "criticality" as task priority. Wolf et al. [6] propose two predictive scheduling algorithms, LAP and EFQ, both of which are in nature based on list scheduling. The basic idea of randomized mapping is to randomly choose a valid mapping and evaluate its performance and repeat this process certain times. [7] and [8] present randomized mapping algorithms with different models for performance evaluation. Genetic algorithm maintains a population of candidate solutions that evolves over time and ultimately converges. Yan et al. [2] generate the initial population by utilizing Monte Carlo method. However, the above algorithms made a lot of assumptions and simplifications:

- **Assumptions of PE architecture.** PEs are supposed to be homogeneous, i.e., execution time of a task processed on different PEs are the same; PEs

are supposed to be linked as pipelining; PEs are supposed not to contain hardware multi-threads. Actually these assumptions are not true in advanced NPs' hardware architecture.
- **Simplification of task partition.** Existing algorithms usually choose pipelining tasks, i.e., except the beginning task and the ending task, each task has and only has one predecessor and one successor. This method can not take full advantage of parallelism of NPs. Describing the network application with DAG is more natural. It reflects characteristics of classification, synchronization, and parallelism of data-packets processing.
- **Simplification of scheduling trigger.** Scheduling occasions can be classified into static scheduling, dynamic scheduling, and partial dynamic scheduling [9]. Partial dynamic scheduling is the trade-off of the former two, in which partial tasks are assigned off-line, and others at run-time. It has low computation cost and can achieve local optimal solution at least.

Our work differs from and perhaps outperforms previous work in several aspects. First, the NP platform is different. We use the advanced hardware architecture, which is heterogeneous, fully-connected, and multi-threading. Second, we adopt partial dynamic mapping, which has been rarely studied in existing NP scheduling. Third, although the strategy of task replication has been deeply studied in cluster systems in the context of scientific computing [10], the task model is very different from ours. Furthermore, we are the first to combine task replication and partial dynamic mapping in NP scheduling.

3 RPDS Algorithm

3.1 Problem Formalization

The scheduling problem is expressed by 5-tuple as following:

$$\Pi = (\boldsymbol{G}, \boldsymbol{D}, \boldsymbol{P}, \boldsymbol{\Theta}, \boldsymbol{\Omega}) \quad . \tag{1}$$

$\boldsymbol{G} = (\boldsymbol{T}, \boldsymbol{E})$ is the dependent relationship graph of tasks, which is usually a DAG. It takes elements in $\boldsymbol{T}$ as nodes, and elements in $\boldsymbol{E}$ as directed edges. $\boldsymbol{T} = \{T_1, T_2, \ldots, T_m\}$ is the set of tasks partitioned from the application. $\langle T_i, T_j \rangle \in \boldsymbol{E}\,(i, j = 1, 2, \ldots, m)$ denotes that T_j is processed after T_i, and there is data transferring from T_i to T_j.

$\boldsymbol{D} = \{D_1, D_2, \ldots, D_l\}$ describes the characteristics of data-packets being processed, such as arrival time and which type of process is needed. $D_i = \langle t_i, G_i \rangle\,(i = 1, 2, \ldots, l)$, t_i is the arrival time of the packet, and G_i is a sub-graph of $\boldsymbol{G}$, i.e., the packet needs to be processed by partial or all of the tasks.

$\boldsymbol{P} = \{P_1, P_2, \ldots, P_n\}$ is the set of PEs in the system. Each PE can load several tasks with each one costs time t_{l}. Each PE has r hardware multi-threads. Every two PEs can communicate directly with the delay of time t_{c}, and communicating delays of inner-PE are ignored.

$\boldsymbol{\Theta}$ is an $m \times n$ matrix. An element of it θ_{ij} denotes the execution time of the task T_i on the PE $P_j\,(i = 1, 2, \ldots, m; j = 1, 2, \ldots, n)$.

$\boldsymbol{\Omega}$ is an $m \times n$ matrix. An element of it ω_{ij} denotes the number of the task T_i on the PE $P_j (i = 1, 2, \ldots, m; j = 1, 2, \ldots, n)$.

The scheduling problem of DAG network processing tasks to heterogeneous multiprocessors is, given input $\boldsymbol{G}$, $\boldsymbol{D}$, $\boldsymbol{P}$, $\boldsymbol{\Theta}$, to get output $\boldsymbol{\Omega}$ and achieve the optimal goal of the system.

3.2 Processing Model

We present the abstract models for network device and processing. Two definitions are presented at first:

Definition 1. *$D_i.RET$, the remaining execution time of data-packet D_i.*

Definition 2. *$D_i.RCT$, the remaining communication time of data-packet D_i.*

Task Model. We assume that the application has been decomposed to $\boldsymbol{T}$ appropriately, and $m < n \times r$, i.e., the number of tasks is less then the total number of threads. One node may have multiple successors, which means the application has conditional branches. For example, the sub-graphs of $\boldsymbol{G}$ are $G_1, G_2, \ldots, G_7$, which represent the processing paths of all types of data-packets (see Fig. 1).

Thread Model. Each thread has its own data buffer. When multiple packets arrive, they are organized as a FIFO queue in the buffer (see Fig. 2).There are four states, unoccupied, blocked, running, and ready, of a thread, which can be transited from one to another in a certain condition.

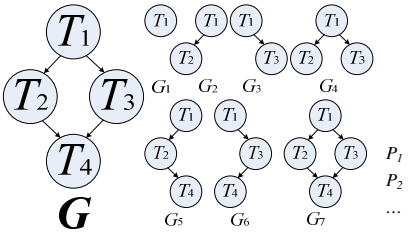

Fig. 1. An example of sub-graphs

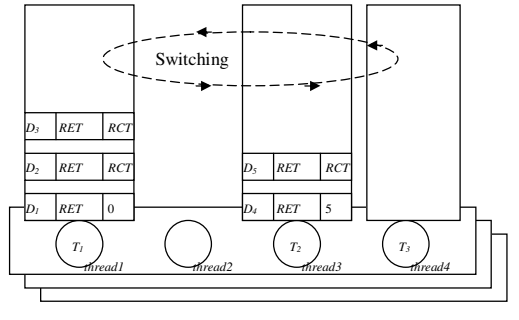

Fig. 2. The architecture of thread buffer

State Model of PEs. The load state of the PE P_k can be busy, normal, or idle. For a period of detecting time t_{d}, the summation time when P_k is running is t_{e}. Then the utilization rate of this PE is:

$$P_k.UR = \frac{t_{\mathrm{e}}}{t_{\mathrm{d}}} \ . \tag{2}$$

Given load upper limit λ_1 and load lower limit λ_2, if $P_k.UR \geqslant \lambda_1$, P_k is busy; if $Pk.UR < \lambda_2$, P_k is idle; P_k is normal when $P_k.UR$ falls between λ_1 and λ_2.

Communicating and Processing Model. For modern processor, communicating is independent with processing. RCT of all data-packets in the buffer minus 1 till 0 after every time unit; RET of the first data-packet in the queue minus 1 till 0 after every time unit if its RCT is 0, while RET of other data-packets remain unchanged; if $D_i.RET$ and $D_i.RCT$ are both 0, the data-packet D_i is finished on this task, and is passed to the tail of successor task queue, resets $D_i.RET$ as θ_{jk} (T_j is the successor task, and P_k is the processor that T_j is loaded), $D_i.RCT$ as t_{C} (different PE) or 0 (the same PE). For a particular task T_j, the buffer of corresponding thread contains data-packets D_1, D_2, ..., D_s. We define $T_j.EEF$ as the earliest expected finish time of T_j:

$$T_j.EEF = \sum_{i=1}^{s} (D_i.RET + D_i.RCT) \quad . \tag{3}$$

The value of EEF implies how busy the task is. For all tasks at the moment, the task whose EEF value is the biggest is the *bottleneck task*.

3.3 Algorithm Procedure

Cost function is used to measure the fitness of mapping results, which is the key of list scheduling algorithm. To take into account execution time, load balance and communication overhead, the cost function is defined as follows:

$$F = a\times\sum_{j=1}^{n}\sum_{i=1}^{m}\omega_{ij}\theta_{ij} + b\times\frac{1}{n}\sum_{j=1}^{n}\left(\sum_{i=1}^{m}\omega_{ij}\theta_{ij} - \frac{\sum_{j=1}^{n}\sum_{i=1}^{m}\omega_{ij}\theta_{ij}}{n}\right)^2 + c\times\sum_{j=1}^{m}\sum_{i=1}^{j}\beta_{ij}t_{\mathrm{C}} \quad . \tag{4}$$

where $\beta_{ij} = 1, \text{if}\,(\langle T_i, T_j\rangle \in \boldsymbol{E}\ \&\ \forall k, \omega_{ik}\times\omega_{jk} = 0)$; or 0, else. The every addend respectively means the linear sum of all execution time, the variance of the execution time of every PE, and the linear sum of all communication delay. a, b, c are corresponding weights. The main procedure of RPDS algorithm is described as follows:

- **Static phase scheduling.** At the initialization of NPs, all tasks are organized as an ordering list. First, calculating the difference between the shortest execution time and the hypo-shortest execution time of each task, the larger the difference is, the higher the task priority is. Then for the task with the highest priority in the list, a PE among those contain unoccupied threads is selected to make the value of cost function F minimal, to which the task is allocated. This process is repeated until all tasks are assigned. At the end of this step the result $\boldsymbol{\Omega}_0$ is obtained. As long as the number of tasks m is less than the total number of threads $n \times r$, each task can occupy a corresponding thread. Apparently there are some redundant unoccupied threads after static phase scheduling, which will be fully utilized at the next phase.
- **Dynamic phase scheduling.** During the run-time of NPs, the PEs' states are detected every t_{d} time. If all PEs are busy, it's unable to adjust, and if all PEs are idle, it's unnecessary to adjust. Therefore when there are some

busy PEs and some idle PEs, the bottleneck task found in busy PEs are duplicated. For each idle PE, the value of cost function F is calculated when the bottleneck task replication is loaded on it, and the PE that makes F the minimal is finally chosen. If all threads on the idle PE is occupied, the task whose EEF is 0 is removed. The pseudo code is given in Fig. 3.

Input: $\boldsymbol{G}, \boldsymbol{\Theta}, \boldsymbol{\Omega}_0, time, P_k.UR, T_j.EEF (k = 1, 2, \ldots, n, j = 1, 2, \ldots, m)$
Output: $\boldsymbol{\Omega}_{time}$

```
While t_d|time & ∃P_k.UR ⩾ λ1 & P_k.UR < λ2
    T_bottleneck ← T_j : max{T_j.EEF}
    For each P_k.UR < λ2
        If all threads in P_k are occupied
            If ∃T_j.EEF = 0 ∉ Ω_0 in P_k
                remove T_j
            Else
                continue
        load T_bottleneck
        calculate F_k
    If more than one F_k
        P_chosen ← P_k : min{F_k}
        P_k except P_chosen roll back //remain not changed
    Return Ω_time
End While
```

Fig. 3. The pseudo code of dynamic phase scheduling

4 Performance Evaluation

4.1 Evaluation Metrics

We use average delay and average throughput as metrics to evaluate the algorithm. For each data-packet D_i, its arrival time is $D_i.t_{\text{receive}}$, and its finished time is $D_i.t_{\text{finish}}$, then the delay of this data-packet is $D_i.Delay = D_i.t_{\text{finish}} - D_i.t_{\text{receive}}$. The average delay and throughput of l data-packets is:

$$Average\_Delay = \frac{1}{l}\sum_{i=1}^{l} D_i.Delay \ . \tag{5}$$

$$Average\_Throughput = \frac{l}{D_l.t_{\text{finish}} - D_1.t_{\text{finish}}} \ . \tag{6}$$

These metrics can work only if $\boldsymbol{G}$, $\boldsymbol{D}$, $\boldsymbol{P}$, and $\boldsymbol{\Theta}$ are the same.

4.2 Simulation Tool

We developed a simulation tool called *dbma*, which implemented the processing model presented in Sect. 3.2. The input was a configuration file in which the

task graph ($\boldsymbol{G}$), packets sending sequence ($\boldsymbol{D}$), PEs ($\boldsymbol{P}$), execution times ($\boldsymbol{\Theta}$), and other parameter values were specified. The outputs included the arrival and finished time of all data-packets (t_{receive}, t_{finish}), the earliest expected finish time of all tasks at each detecting time (EEF), the utilization rates of all PEs at each detecting time (UR), and all mapping results ever have ($\boldsymbol{\Omega}$). Uniform virtual time unit was used in simulation.

Specially, to specify $D_i = \langle ti, Gi \rangle$ for every data-packet separately is time-consuming because there are thousands of packets in the experiment. For t_i, we assumed that the packet sending intervals follow the exponential distribution. For G_i, we added probabilities of data-packets transferred from T_i to T_j. The execution time ranged from 0 to 100, whereas 0 means that the task is not executable on that PE.

4.3 Experimental Results

The choice of parameters is important to the experiments. We used some parameters as default (see Table 1, and varied others to observe their effect to performance.

(1) Sending Data-Packets at Constant Speed. The DAG in this experiment is presented in Fig. 1, where $m = 4, n = 4, r = 3$. The default branch probability is 1. The execution time of each task is shown in Table 2. To verify the performance of RPDS, we implemented several variations (i.e., different types) of it, which are presented in Table 3.

Table 1. Default parameters

Parameter	t_{C}	t_{l}	t_{d}	λ_1	λ_2	a	b	c
Value	10	20	500	0.9	0.7	0.5	0.1	0.4

Table 2. Execution time

	P_1	P_2	P_3	P_4
T_1	86	57	68	85
T_2	29	61	52	18
T_3	0	53	16	68
T_4	94	6	15	86

Table 3. RPDS algorithm variations

Type ID	Schedule Multi-DAGs at static phase	Have dynamic phase	Allow synonymous tasks in one PE
2	No	Yes	No
4	No	No	No
6	Yes	Yes	No
8	Yes	No	No
10	No	Yes	Yes

Sending data-packets at the constant speed every time unit respectively (e.g., 0.01 denotes that the average sending interval is 100 time units), the results are shown in Fig. 4 and Fig. 5.

We can observe that the average delay and average throughput are both increasing along with the increase of the constant speed. The performance of RPDS variations ranks as: Type-10 > Type-2 > Type-6 > Type-8 > Type-4. Let the delay bound be 1500, we can see that the throughput rate (throughput / sending speed) keeps 1. If exceeding this bound, the throughputs do not increase

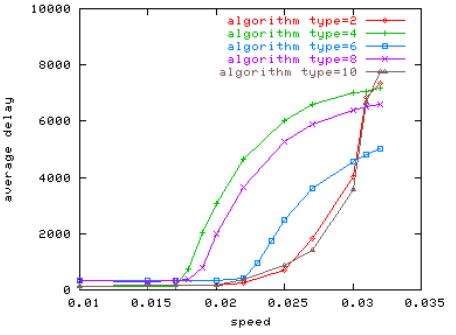

Fig. 4. Packet speed vs. delay

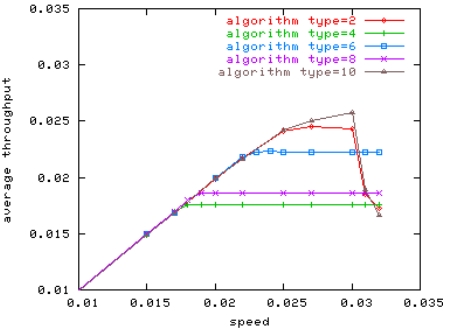

Fig. 5. Packet speed vs. throughput

any more but have trends to decrease. When the packet sending speed is lower than 0.017, the five variations are grouped into two classes: Type-2, Type-4, and Type-10 vs. Type-6 and Type-8. The average delays of the former are about 50% lower than those of the latter. But as the speed increasing, the two classes turn to be: Type-2, Type-6, and Type-10 vs. Type-4 and Type-8. The largest acceptable speeds of the former are 30% larger than those of the latter.

That is to say, when the workload is light, scheduling without static replication is superior to that with static replication; while the workload is heavy, scheduling with dynamic replication performs better than that without dynamic replication. The reason is that there is no need to duplicate at light workload, and furthermore the delay is increased after replication because of the frequent transferring between PEs of data-packets. Dynamic replication abated the pressure of the bottleneck task effectively at heavy workload, balanced the tasks among PEs, and therefore increased the throughput.

(2) Sending Data-Packets at Variable Speed or Probabilities. In this experiment, the DAG, execution time, and algorithm types are the same as those in experiment (1). The packet sending speed and branch probabilities are different, which are shown in Table 4.

Table 4. Packet sending speed; Branch probabilities of $T_1 \rightarrow T_2/T_1 \rightarrow T_3$

Time	0–10000	10000–20000	20000–30000	30000–40000	40000-50000
Speed	0.017	0.0185	0.0195	0.0235	0.026
Time	0–15000	15000–30000	30000-		
Probability	0.4/0.4	0.1/0.9	0.9/0.1		

We selected Type-10 (with dynamic replication) and Type-8 (without dynamic replication) to compare with each other. The results are shown in Fig. 6 and Fig. 7.

This experiment shows the reason why RPDS can reduce the delay and improve the throughput in more detail. The delay of packets in Type-10 algorithm

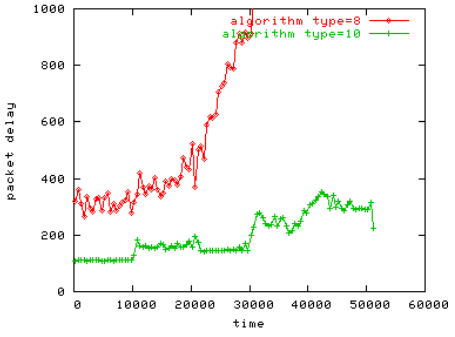

Fig. 6. Different speed vs. delay

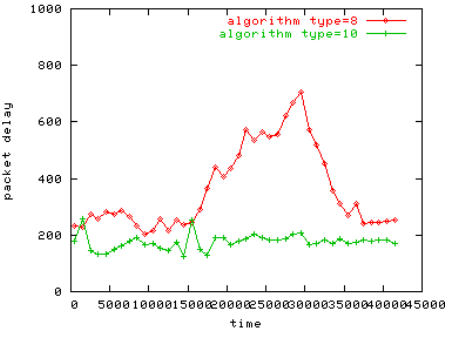

Fig. 7. Different probabilities vs. delay

reaches a small peak after changes of traffic characters (speed or probabilities), and turns to be smooth soon. But for Type-8, the delay changes dramatically according to the changes of traffic(see Fig. 8 and Fig. 9). It is obvious that the detection and adaptation of RPDS contribute to the performance.

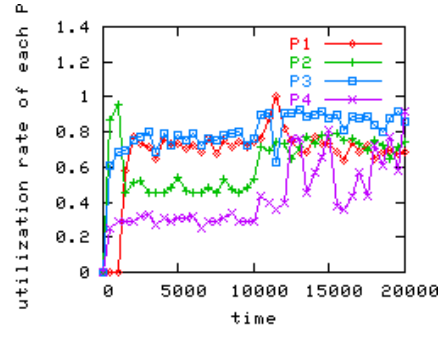

Fig. 8. Utilization rates of PEs

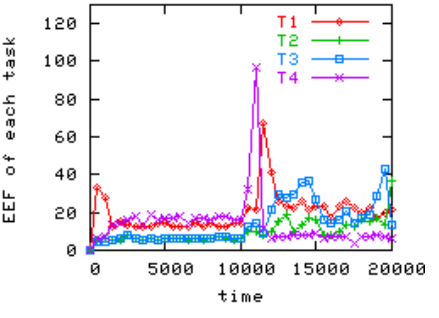

Fig. 9. *EEF* of tasks

5 Conclusions

The Replication-Based Partial Dynamic Scheduling (RPDS) is proposed in this paper. It tries to solve the problem of processing resources scheduling on the heterogeneous, fully-connected, and multi-threading NP hardware architecture. The main idea of RPDS algorithm is two-phase scheduling: static phase and dynamic phase. The static phase scheduling performs task pre-assignment before processing data-packets. It guarantees that each task could hold the minimal processing resources and keep the cost lowest. The dynamic phase scheduling occurs during the processing data-packets. When busy PEs and idle PEs coexist, the bottleneck task will be duplicated to the idle PE which makes the lowest cost. The future work includes the theoretic verification of RPDS, update of *dbma*, and experiments to the default parameters, etc.

Acknowledgments. This work is partially supported by Graduate Starting Seed Fund of Northwestern Polytechnical University (No. M016634).

References

1. Malloy, B.A, Lloyd, E.L., Soffa, M.L.: Scheduling DAG's for Asynchronous Multiprocessor Execution. IEEE Trans. Parallel and Distributed Systems 5(5), 498–508 (1994)
2. Yan, S., Zhou, X., Wang, L., Wang, H.: GA-Based Automated Task Assignment on Network Processors. In: ICPADS 2005. Proc. of the 11th international Conference on Parallel and Distributed Systems, July 20–22, 2005, pp. 112–118. IEEE Computer Society Press, Los Alamitos (2005)
3. Franklin, M., Datar, S.: Pipeline task scheduling on network processors. In: Proc. of Third Network Processor Workshop in conjunction with Tenth International Symposium on High Performance Computer Architecture (HPCA-10), pp. 103–119 (February 2004)
4. Yang, L., Gohad, T., Ghosh, P., Sinha, D., Sen, A., Richa, A.: Resource mapping and scheduling for heterogeneous network processor systems. In: ANCS 2005. Proc. of the 2005 Symposium on Architecture for Networking and Communications Systems, pp. 19–28 (2005)
5. Ramaswamy, R., Weng, N., Wolf, T.: Application Analysis and Resource Mapping for Heterogeneous Network Processor Architectures. In: Proc. of Network Processor Workshop, Madrid, Spain, pp. 103–119 (2004)
6. Wolf, T., Pappu, P., Franklin, M.A.: Predictive scheduling of network processors. Comput. Networks 41(5), 601–621 (2003)
7. Weng, N., Wolf, T.: Pipelining vs. Multiprocessors-choosing the Right Network Processor System Topology. In: Proc. of ANCHOR 2004, Munich, Germany (2004)
8. Weng, N., Wolf, T.: Profiling and mapping of parallel workloads on network processors. In: Proc. of 20th ACM Symposium on Applied Computing (SAC) (March 2005)
9. Wolf, T., Weng, N., Tai, C.: Design considerations for network processor operating systems. In: ANCS 2005. Proc. of the 2005 Symposium on Architecture for Networking and Communications Systems, October 26–28, 2005, pp. 71–80. ACM Press, New York (2005)
10. Aggarwal, A., Franklin, M.: Instruction Replication for Reducing Delays Due to Inter-PE Communication Latency. IEEE Trans. Comput. 54(12), 1496–1507 (2005)