Mapping Parallel Workload onto Network Processors Using Simulated Annealing

Satish Dechu[1], Ning Weng[2], and Benfano Soewito[2]
[1]Sun Microsystems, Inc., Santa Clara, California, USA
[2]Department of Electrical and Computer Engineering,
Southern Illinois University, Carbondale, Illinois, USA

Abstract

Network Processors (NPs) are promising components to build a performance-scalable and function-flexible network systems. A clear trend has been observed that more and more NPs employ multiple simple processors to run multiple packet processing applications in parallel. The key challenge for application developers is how to program NPs for high performance. This paper presents an automated task scheduling technique based on Simulated Annealing(SA). By incorporating tasks dependency into scheduling list, SA can quickly remove the invalid mappings and evolve to the high quality solutions. Particularly, a transition probability is defined to help converge, which is depending on system throughput and control parameter 'temperature'. Our mapping technique is able to take advantage of task-level and application-level parallelism to maximize system performance. The simulation results show that this proposed technique can generate high quality mapping comparing to other heuristics by mapping some sample network applications.

Keywords

Multiprocessors, Network Processors, Application Mapping, Embedded Systems, and Simulated Annealing [1]

1 Introduction

Network Processors (NPs) are embedded system-on-chip multiprocessors that are optimized to perform simple packet processing tasks at data rates of several Gigabytes per second. They are the key components to build a performance-scalable and function-flexible network systems. To meet the performance demands of increasing link speeds and more complex network applications, NPs are implemented with several dozen of processor cores and run multiple packet processing applications in parallel. Commercial examples of NPs are numerous [2, 4, 6].

Programming of such heterogeneous multiprocessor systems is difficult, as the overall performance depends on the fine-tuned interaction of different system components (processors, memory interfaces, shared data structures, etc.). The main problem is handling the complexity of various, interacting NP system components, which include multiple multithreaded processing engines, different types of on- and off-chip memory, and a number of special-purpose co-processors. This problem is much more difficult than programming of conventional workstation or server systems, because the complexities of these systems are hidden by the operating system or do not express themselves as drastically due to their uniprocessor architecture.

Network processing is inherently a dynamic process, which means easily reprogramming is essential. The main motivation using network processor (rather than in a faster, more power-efficient custom logic device) is the need to change the networking functionality over time. Changing traffic patterns, new network services and protocols, new algorithms for flow classification, and changing defenses against denial of service attacks present the dynamic background that a programmable router needs to accommodate. This requires that the router (1) can implement multiple packet processing applications at the same time, (2) can quickly add and remove processing functions from its workload, and (3) can ensure efficient operation under all circumstances. In particular, the management of various system resources is important to avoid performance degradation from resource bottlenecks.

To simplify complexity of this programming , a number of domain-specific programming languages and optimizing

[1]This work was performed while at the Southern Illinois University Carbondale.

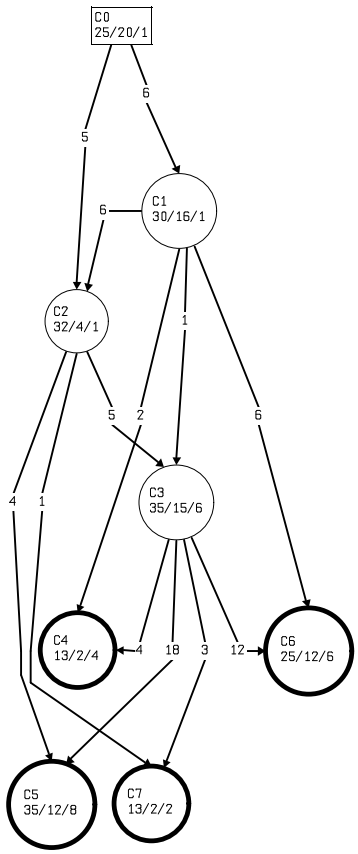


Figure 1. A Sample of Network Applications in Annotated Directed Acyclic Graphs.

compilers are currently being developed [5, 13, 14]. Most of these approaches aim at optimizing a single application (i.e., router functionality) statically for the underlying hardware. In current network processors, most performance critical tasks are implemented and fine-tuned in assembly (e.g., to balance the processing times in each step of a software pipeline). As a result, slight changes in the functionality can have drastic performance impacts which require re-tuning. Due to the necessary fine-tuning of individual application, it is difficult to integrate and dynamically change multiple packet processing functions on a single network processor.

Assuming network application (workload) can be represented as a task graph [12], then the key question is how to map a task graph onto highly parallel NPs efficiently. To solve this NP-hard mapping problem [10], in this paper we introduced "Simulated Annealing" algorithm which can solve the problem of achieving global optimum mapping which gives the best throughput of the NP-system.

The remainder of this paper is organized as follows. We formalize the problem of mapping application task graphs onto NPs in Section 2. Related work is discussed in Section 3. Section 4 describes our simulated annealing used to map network processor workload. Results are presented and discussed in Section 5. Section 6 summarizes and concludes the paper.

2 Problem Formulation

Packets from network are classified and routed by a scheduler to one specific processing function, named application. This application can be represented by a annotated directed acyclic graph (aDAG): $G = (V, E)$, where V is a set of nodes and E is a set of directed edges. A node in the aDAG represents a processing task and an edge shows the precedence constraints among the nodes. The 3-tuple value of a node i shows number of instruction p_i, number of memory reads r_i and number of memory writes w_i. The weight of edge e_{ij} in the aDAG corresponds to the communication cost. Figure 1 shows a sample of network applications in aDAGs. The annotations in a node are the node name and a 3-tuple (processing/reads/writes). The weight on the edges is the number of data and control dependencies between nodes.

Figure 2 shows our parameterized NPs, which consists of four components: processing elements (PE), shared interconnects (IC), memory interfaces (MI), and hardware accelerators (HA). The key parameters are: the width of the pipeline (W), the depth of the pipeline (D), the number of stages per communication interconnect (I), the number of memory channels shared by a stage of processing elements (M), and the number of hardware accelerators (H) per stage. These parameters enable us to represent a wide range of possible NP architectures: parallel multiprocessor when D equals to 1; pipelined architecture when W equals to 1; and hybrid architecture. A hybrid architecture when set W, D, I=1...D, and M=1...W to any combination of values.

The system throughput is determined by maximum latency of pipeline and the number of parallel processing tasks n_{ADAG}. The latency of pipeline considers of processing, inter-processor communication, contention on memory interfaces, and pipeline synchronization effects. With a system clock rate of clk, the throughput of the system can be expressed as

$$thx = \frac{n_{ADAG} \cdot clk}{\max_{i=1}^{D} \left(\tau_{comm_i} + \max_{j=1}^{W} \left(\tau_{proc_{i,j}} + \tau_{mem_{i,j}}\right)\right)} \quad (1)$$

Thus the goal of NPs mapping is to maximize number of aDAGs but minimize the maximum latency of pipeline, because overall system performance is evaluated how many packets (it equals to the number of aDAGs mapping to NPs) are processed during the pipeline stage time. During this procedure, constraints should be considered, such as limiting instruction memory of each PE and dependency of tasks in each aDAG.

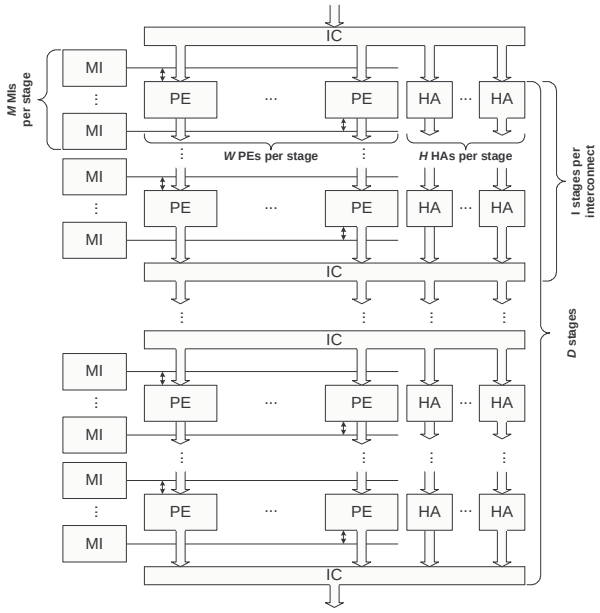
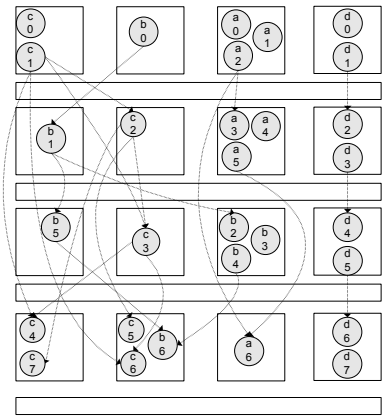
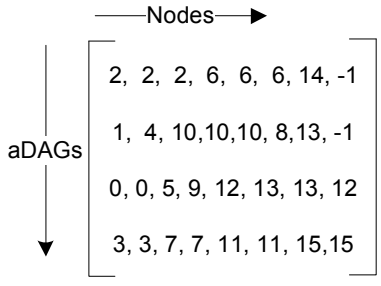


Figure 2. Generalized NP Architecture.



(a)



(b)

Figure 3. Sample Mapping of aDAG onto NPs and Coding of the Mapping.

3 Related Work

Recent work in the area focuses on applying higher-level programming abstractions to simplify code development (e.g., domain-specific programming language by Teja [14], C compiler for IXP by Intel [3]). Other programming models include NP-Click proposed by Shah et al. [13], by extending to the Click modular router [8]. The major drawback is requiring the application developer an in-depth understanding of the NP system architecture. Ostler and Chatha [11] presented an automatic ILP-based technique for application mapping on NPs, however, the effectiveness of this technique relies on accurate characterizing a process latency and its data size.

Mapping algorithms for assigning task graphs to multiprocessors [9, 1] is conceptually similar to mapping tasks inside an NP, but there are significant differences in the underlying system architecture. [15] describe a simple randomized algorithm used in run-time system, which does not have sufficient resource to execute complicated mapping algorithm. However, this algorithm is not scalable for large space exploration because significant time is spent filtering out invalid mapping that violate dependency violations. [16] present a heuristic which combines Genetic Algorithm (GA) and greedy approach to programming a pipelined NPs. However, their work does not consider memory access time and communication cost, which can be the bottleneck of overall system performance.

4 Mapping Algorithm

Our heuristic solution to the mapping problem is based on Simulated Annealing [7]. The key technique is inspired from annealing in metallurgy. By analogy with this physical process, our SA algorithm replaces the current best mapping with a random solution chosen with a probability which is depending on system throughput and control parameter 'temperature'.

4.1 Coding of Mapping

Figure 3 (a) shows an instance of a mapping result of four different aDAGs on a topology with a depth of four stages and a width of four processing elements. The square boxes represent the processing elements (PE), the circles the nodes of the aDAGs and the horizontal bar the communication channel. The arrows indicate the dependency of aDAG nodes. This mapping is for illustration purpose, not necessarily the optimal one.

The coding of m aDAGs mapped to q PEs can be represented by two dimension array S(m,n):

$$S(m,n) \quad = \quad \begin{pmatrix} s_{11} & s_{12} & \cdots & s_{1n} \\ s_{21} & s_{22} & \cdots & s_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ s_{m1} & s_{m2} & \cdots & s_{mn} \end{pmatrix} \quad (2)$$

```
1  Migration(Map)
2  begin
3      a ← ADAG in Map;
4      n ← select random node(1,|a|);
5      p_i ← RandomProc(Map);
6      migrate n to p_i;
7      return Map;
8  end
9  Exchange(m, f)
10 begin
11     randomly choose ADAG a in workload;
12     index ← Select rand(1,|a|);
13     for i ← 0 to index do
14         O_1(i) = m(i);
15         O_2(i) = f(i);
16     end
17     for i ← index to |a| do
18         O_1(i) = f(i);
19         O_2(i) = m(i);
20     end
21     return(O_1,O_2);
22 end
```

Algorithm 1: Key Operations of SA

where n is the maximum number of nodes among all aDAGs. There is one to one correspondence between task node t of aDAG a and a string. The value of each string is the index of PEs, onto which node t of aDAG a is mapped.

Figure 3 (b) shows the representation of sample mapping in Figure 3 (a). In these mapping, m –number of aDAGs equals to 4; n –maximum number of nodes among all aDAGs is 8 and q–number of PEs is 16, thus the mapping can be represented by a 4x8 array. Each row of this array represents the mapping of one aDAG, for example, in row 2 $(1, 4, 10, 10, 10, 8, 13, -1)$ indicates the mapping of aDAG b in Figure 3(a). Each element of array indicates the index of PE, onto which node t of an aDAG is mapped. The index of one PE p_{rw} is $(r * W + w)$, where W is width of NPs pipeline. For instance, node 2 of aDAG b, $b2$ is mapped onto NPs in row 2 and column 2, with $W = 4$, its string is 10. Should note, the -1 is to indicate that there is no node in this position.

Assuming there are no dependency among aDAGs – if there are constraints between two aDAG, we can merge two aDAG into one, therefore we can schedule each aDAG individually and independently. This independency of aDAG allows us to perform migration and exchange separately for each aDAG, for example cross point for aDAG a is not necessary same to that of aDAG b. The detail of algorithm is discussed in the following.

4.2 Simulated Annealing

Simulated Annealing (SA) is imitating nature's process of annealing in order to find the approximate solutions to combinatorial optimization problems. In SA, the throughput increment is governed by a cooling temperature $temp$ which varies from given high value to a low value slowly. At every temperature, a fixed number of minimization steps are tried. At every minimization step, a new mapping is chosen and throughput calculated. If new mapping throughput is better than old mapping throughput then the new mapping is accepted at that step. If the throughput is less than previous mapping then the new mapping is accepted with a probability by the following equation:

$$\rho_{accep} = e^{\frac{-\delta thx}{\kappa \cdot T}} \quad (3)$$

Where κ is the Boltzman's constant, T is the control parameter 'temperature' and δthx is the difference of system throughput. This accepting probability prevents the method from becoming stuck in a local minimum.

```
1  SaAnnealing(Map)
2  begin
3      P_1 ← InitialMapping();
4      P_2 ← InitialMapping();
5      bestMap ← best among P_1,P_2;
6      C_0 ← Performance(bestMap);
7      while T > T_f and L > 0 do
8          L ← RunsAtGivenT;
9          migorex ← rand();
10         if migorex < 0.5 then
11         begin
12             Map ← (P_1 orP_2);
13             NewMap ← Migration(Map);
14             if (Performance(NewMap) >
                   Performance(Map) or rand() < prob)
               then
15                 bestMap ← NewMap;
16                 P_1 or P_2 ← NewMap;
17             end
18         end
19         else
20         (O_1,O_2)←Exchange(P_1,P_2);
21         if (Performance(O_1 or O_2)
           >Performance(bestMap)) then
22             bestMap ← (O_1 or O_2);
23         end
24         (P_1, P_2)← SelectTopTwo(P_1,P_2,O_1,O_2);
           L−−; T = K × T;
25     end
26     return bestMap;
27 end
```

Algorithm 2: Main Procedure of SA

The SA starts from stage of "InitialMapping" in Algorithm 2 (lines 3–4) where initial individuals are randomly generated. Initially the temperature is assigned a value To. and then the temperature is decreased in steps by forcing

Table 1. System Parameters and Its Values.

Item	Symbol	Description	Default
topology	d	depth of NPs topology	16
	w	width of NPs topology	16
	i	# of stage per interconnect	1
	m	# memory channel/stage	2
	h	# accelerator/stage	0
processor	f	processor clock freq.	600Mhz
	c	context switching overhead	1
	t	# of threads per processor	1
memory	l	size of insturciton store	8K
	s	memory service time	10
application	n	# ADAGs to be mapped	100
SA	g	# of generations	1000
	T	initial temperature	10000
	$alpha$	cooling constant	0-1
	k	Boltman's constant	0.95-0.99
	$p_{exchange}$	probability of exchange	0.5
	$p_{migration}$	probability of migration	0.5

$T = \alpha \times T_i$. Where α is the cooling constant $0 < \alpha < 1.0$. Ideally there should be infinite number of trials at every temperature. However in practice the 'runs at a given temperature' number is a finite number. This experiment is continued until temperature is 0.

The main procedure is "Simulated Annealing" in Algorithm 2 (lines 1–27), which provides inputs to three subprocedures and calls them. Each mapping is evaluated using our analytical performance model. If the new mapping is a better solution in terms of the optimization metric, it is recorded for comparison to future solutions. At the end of the mapping process, the best overall mapping is reported.

Initially two parent mappings are generated by mapping all ADAGs randomly on NP-topology. In each trial step either migration or exchange operations are performed to get the new mapping. In migration, Algorithm 1 (lines 1–8), a processor is chosen randomly and also a random node in a given ADAG which migrates to another processor so that the dependency maintains. The exchange operation, Algorithm 1 (lines 9–22), selects a random ADAG from two parent mappings and generates two children mappings by exchanging nodes between them by using random partition.

5 Results

We present and analyze several results obtained by utilizing the algorithm discussed in Section 4 for mapping representative network processing applications onto our general NP architecture in Figure 2. First, we compare the mapping results with different matching algorithms to illustrate the effectiveness of SA. Then, we evaluate the sensitivity of system performance to different SA variables. Finally, we present results showing the impact of applications parallel processing on system performance. Table 1 lists key system parameters and their description. Unless otherwise specified, default values are used to generate results.

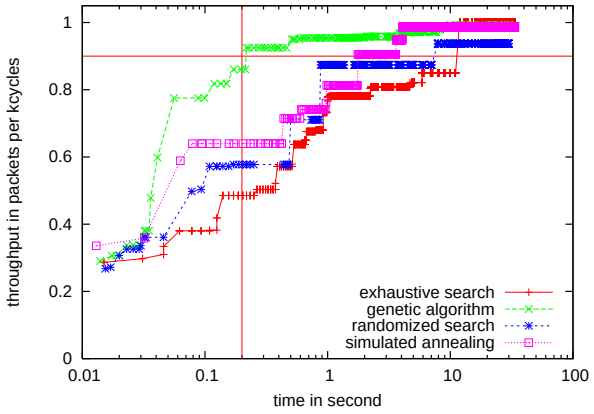


Figure 4. Mapping Quality Comparison between Algorithms.

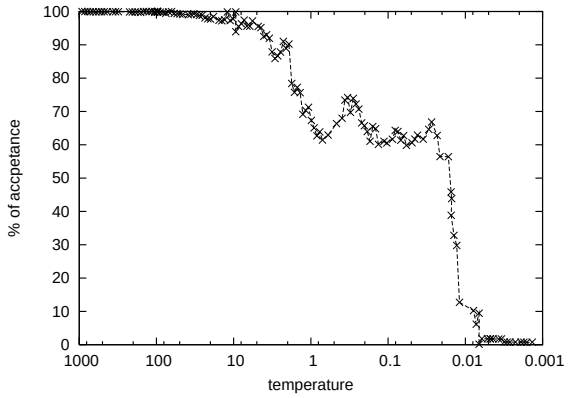


Figure 5. Percentage of probability of acceptance value at each iteration in *10N*

5.1 Mapping Evaluation

SA is a promising approach to solve the NP complete problem, however we would like to evaluate how well this approach works in terms of mapping accuracy and efficiency by comparison to other search algorithms: Exhaustive Search, Randomized Search and Genetic Algorithm.

Exhaustive Search finds the best solution by trying every possibility is known as an exhaustive search, direct search, or the brute force" method. The basic idea of exhaustive search is enumerating all the possible mappings and then evaluating the performance of each valid one. For the topology with p processing elements, the ADAG to be mapped consisting of n nodes, with m total same ADAGs to be mapped, the total number of possible mappings is p^{mn}. Here our experiment uses pipelined topology (d=2, w=2) and the number of ADAGs to be mapped is 3 and each ADAG consists of 5 nodes. The best mapping from Exhaustive Search will be upper bound of the system.

Randomized Search achieves a good approximation to

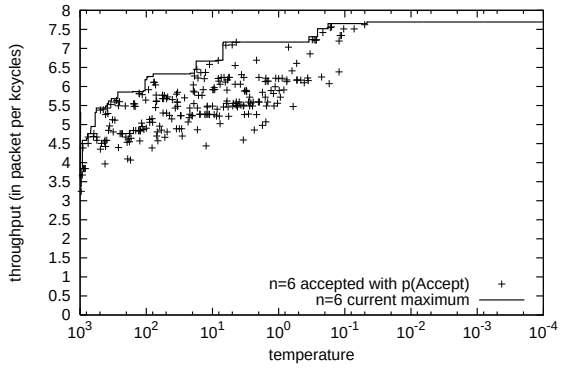


Figure 6. Current maximum throughput and worse mappings accepted with *p(Accept)* at each iteration in 10N

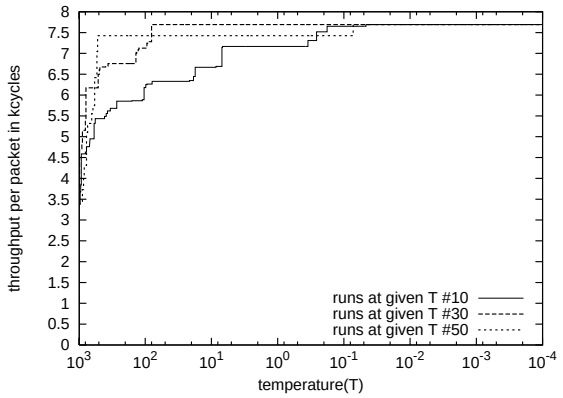


Figure 7. Current maximum throughput at each temperature for different number of runs at given temperature

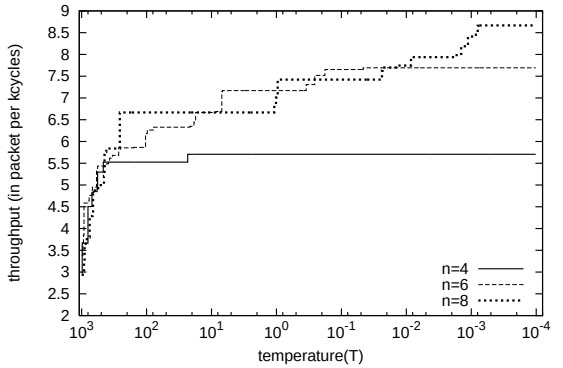


Figure 8. Throughput with respect to temperature for different number of aDAGs (n)

the best solution by randomly choosing a valid mapping and evaluating its performance, and then repeating this process a number of times and picking the best solution that has been found over all iterations. The intuition behind this is that any algorithm that does not consider all possible solutions with a non-zero probability might get stuck in a local optimum. With the randomized approach any possible solution is considered and chosen with a small, but non-zero probability.

Genetic Algorithm (GA) is inspired by evolutionary biology such as inheritance, mutation, selection, and crossover. GA works with a population of individuals, each representing a possible solution to a given problem. The highly-fit individuals are given opportunities to reproduce, by cross breeding with other individuals in the population; the least fit members of the population are less likely to get selected for reproduction, and so die out. A whole new population of possible solutions is thus produced by selecting the best individuals from the current generation, and mating them to produce a new set of individuals. This new generation contains a higher proportion of the characteristics possessed by the good members of the previous generation. The weakness of this algorithm is there is no statistical guarantee that it converges to a optimum solution.

The comparison between different algorithms shown in figure 4. We can observe that SA and GA are converging quicker than random and exhaustive search algorithms. Exhaustive search examines all possibilities of mappings and returns the best one. So the time to converge will increase exponentially even for small increase in problem size. GA is converging faster than SA, but GA depends heavily on population size and it is difficult to determine the size of population which depends on the size of the problem. So there is a huge chance that it could converge to local optimum instead of global one.

As we mentioned above, the complexity of exhaustive search is O(p^{mn}): p is the total number of processors (equals to d*w); mn is the total number of nodes of multiple aDAGs. In Figure 4, we only show a small design with number of processors equaling to 4. The purpose is to illustrate that our SA algorithm does perform well in term of efficiency and convergence.

5.2 SA Variables Fine Tuning

Figure 5 shows how our normalized probability is varying with temperature T. We can observe from this figure, that the value of probability is close to one at initial high temperature levels and decreases exponentially as the temperature goes down and come to '0' at final temperature. Hence at high temperatures, algorithm accepts all mappings even those which are worse compared to current maximum value and it starts rejecting these slowly as temperature and probability is going down and stops at '0' probability value. The algorithm performs some iterations at this stage and ends with optimum throughput value. This figure also

shows the value of probability at each run in *10N* and points which are accepted. In $10N$, 10 represents the number of runs at given temperature and N is the total number of temperature levels from initial temperature to final temperature.

Figure 6 shows the current maximum performance at each temperature and it also shows the worse mappings as points which are accepted to cross the local optimums with probability of acceptance. In this figure we can observe that in the initial iterations algorithm accepts many worse mappings. As temperature is decreasing algorithm slowly decreases the number of acceptances, accepting only those which are close to current maximum and stops at optimum throughput value.The number of applications used here is '6'. Total number of iterations is 10N.

Our algorithm is evaluated as different runs at given temperature of *50N, 30N, and 10N*. Figure 7 shows that as the number of runs increases algorithm reaches to its optimum value very soon. This is because at each temperature level the algorithm will get more chance to find best mapping at that level. But it may not reach the optimum value as lesser number of runs. So we need to try different number of runs to get the most optimum value.

Figure 8 shows improving throughput with temperature for different number of ADAG. The throughput gain is due to multiple application executing in parallel. This Figure shows that SA based mapping method can take advantage of parallel workload to speed up the system throughput.

6 Conclusion

In this work we present a methodology for mapping network applications as a task graphs (aDAGs) onto network architecture. We solve this NP-complete problem with Simulated Annealing. By tuning Simulated Annealing parameters our algorithm converges to optimum solution quickly. The mapping quality is evaluated by an analytical performance model, which captures the key system aspects of a heterogeneous network processor system. The results show the algorithm convergence speed and quality of solution as compared with other algorithms.

We believe that this methodology poses a promising approach to managing the complexities of highly parallel, heterogeneous network processors and embedded systems in general. The mapping can be done entirely automatically from a uniprocessor implementation of an application. It is conceivable that such functionality will become part of software development kits and run-time environments of future network processors.

References

[1] T. M. Austin and G. S. Sohi. Tetra: evaluation of serial program performance on fine-grain parallel processors. Technical Report 1163, Computer Science Department, University of Wisconsin, Madison, WI, July 1993.

[2] C-Port Corporation. *C-5^{TM} Digital Communications Processor*, 1999. http://www.cportcorp.com/solutions/docs/c5-brief.pdf.

[3] J. Dai, B. Huang, L. Li, and L. Harrison. Automatically partitioning packet processing applications for pipelined architectures. *SIGPLAN Not.*, 40(6):237–248, 2005.

[4] EZchip Technologies Ltd., Yokneam, Israel. *NP-1 10-Gigabit 7-Layer Network Processor*, 2002. http://www.ezchip.com.

[5] S. D. Goglin, D. Hooper, A. Kumar, and R. Yavatkar. Advanced software framework, tools, and languages for the IXP family. *Intel Technology Journal*, 7(4):64–76, Nov. 2004.

[6] Intel Corp. *Intel IXP2800 Network Processor*, 2002. http://developer.intel.com/design/network/products/npfamily/ixp2800.htm.

[7] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.

[8] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug. 2000.

[9] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, Dec. 1999.

[10] B. A. Malloy, E. L. Lloyd, and M. L. Souffa. Scheduling DAG's for asynchronous multiprocessor execution. *IEEE Transactions on Parallel and Distributed Systems*, 5(5):498–508, May 1994.

[11] C. Ostler and K. S. Chatha. An ilp formulation for system-level application mapping on network processor architectures. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 99–104, New York, NY, USA, 2007. ACM Press.

[12] R. Ramaswamy, N. Weng, and T. Wolf. Application analysis and resource mapping for heterogeneous network processor architectures. In *Proc. of Network Processor Workshop in conjunction with Ninth International Symposium on High Performance Computer Architecture (HPCA-10)*, pages 103–119, Madrid, Spain, Feb. 2004.

[13] N. Shah, W. Plishker, and K. Keutzer. NP-Click: A programming model for the intel IXP1200. In *Proc. of Network Processor Workshop in conjunction with Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*, pages 100–111, Anaheim, CA, Feb. 2003.

[14] Teja Technologies. *TejaNP Datasheet*, 2003. http://www.teja.com.

[15] T. Wolf, N. Weng, and C.-H. Tai. Design considerations for network processor operating systems. In *Proc. of ACM/IEEE Symposium on Architectures for Networking and Communication Systems (ANCS)*, Princeton, NJ, Oct. 2005.

[16] S. Yan, X. Zhou, L. Wang, and H. Wang. Ga-based automated task assignment on network processors. In *ICPADS '05: Proceedings of the 11th International Conference on Parallel and Distributed Systems (ICPADS'05)*, pages 112–118, Washington, DC, USA, 2005. IEEE Computer Society.