Dynamic Workload Profiling and Task Allocation in Packet Processing Systems

Qiang Wu and Tilman Wolf
Department of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA
{qwu,wolf}@ecs.umass.edu

***Abstract*—Computer networks require increasingly complex packet processing services on routers to adapt to new functionality, security, and performance requirements. Embedded multicore packet processing systems that can provide this capability are difficult to program and manage at runtime. We propose a novel way of representing processing tasks, obtaining runtime profiling information, and mapping tasks to processors. By duplicating processing tasks with heavy processing requirements, a more balanced workload can be obtained. The mapping algorithm considers that balance when assigning tasks to processors as well as the cost of inter-processor communication. Our evaluation results show that our approach can improve the system throughput by 2.39–2.89 times at a cost of 1.49–1.64 times higher inter-processor communication.**

***Index Terms*—High-performance router, packet processing, network processor, scheduling**

I. Introduction

Router systems are the key components that determine the functionality and capabilities of computer networks. With recent efforts to design the architecture of the next-generation Internet [1], packet processing systems have captured the spotlight of networking research. While the original Internet design called for simple store-and-forwarding capabilities on routers, the complexity of today's networks and their performance and security requirements has significantly increased the number of processing steps for IP forwarding on routers [2]. We expect this trend to continue in next-generation networks, where an increasing number of heterogeneous end-system and diverse protocols will need to be supported. The implication for router design is that such diversity in processing demands cannot be implemented in customized logic, but requires *general-purpose packet processing systems*.

To achieve the throughput required for high-performance routers, packet processing is typically performed on highly parallel, embedded multicore systems. So-called "network processors" exploit the inherent parallelism that is present in packets from different network connections and use tens of simple processor cores for various packet processing tasks. One major challenge in such systems is how to program these processors to efficiently process packets and share common resources (e.g., memory, processor interconnect).

When programming multiprocessor systems, there are three fundamental problems that need to be considered:

This material is based upon work supported by the National Science Foundation under Grant No. 0447873.

1) Partitioning of Application into Tasks: The packet processing application needs to be partitioned in order to be distributed across processing resources. This step is also necessary since simply duplicating the entire application across all processor cannot be done efficiently due to limitations of instruction memory size.
2) Mapping of Tasks to Processors: Processing tasks obtained from the partitioning step need to be allocated to processing resources. This mapping has a significant impact on the performance of the system as it determines how effectively resources are utilized and how much contention on shared resources occurs.
3) Dynamic Adaptation of Mapping under Changing Traffic: Adapting the task mapping to changes in the processing requirements due to changes in traffic is particularly important in the networking domain. This step is crucial for efficient operation of packet processing systems, but often neglected.

To address these issues, we present a novel approach to managing task allocation on packet processing systems. The key idea is to instrument processing tasks to obtain dynamic runtime information on workload requirements. The most computationally demanding tasks are replicated across multiple processor cores to ensure that sufficient processing resources are dedicated. Finally, a novel mapping algorithm is used to determine a task mapping that aims for low overhead on the system interconnect. We show the effectiveness of this approach in an evaluation that considers dynamically changing traffic patterns. The specific contributions of our paper are:

- Partitioned Application Representation with Profiling Information. We present a representation of packet processing workloads that considers traffic characteristics, task interdependencies, and processing times. This is the foundation for our task duplication and mapping algorithm.
- Task Duplication Algorithm. To balance tasks according to their processing demands we use a novel task duplication algorithm. The algorithm determines the number of duplications based on profiling information.
- Task Mapping Algorithm. The task mapping algorithm places tasks on processing resources based on a depth-first search to reduce the amount of data that needs to be transferred across the processor interconnect.

The effectiveness of our approach is illustrated through experimental validation.

The remainder of this paper is organized as follows. Section II discusses related work. Section III introduces the application representation, the duplication algorithm, and the mapping algorithm. Evaluation results are presented in Section IV, and Section V concludes this paper.

II. Related Work

The need for increasingly complex packet processing tasks in the data path of routers has been made for current and emerging network architectures. Current IP routers implement a range of packet processing functions that go well beyond simple IP forwarding (e.g., QoS, monitoring, accounting, etc. [2]). Proposals for next-generation network architectures further elevate custom packet processing to a fundamental network capability (e.g., network services [3], end-to-end services [4]). These trends imply that general-purpose processing engines are essential building blocks for the data path of current and future router systems.

Network systems that provide such general packet processing capabilities range from routers with network processors [5] to high-performance implementations of overlay networks [6]. To generalize the hardware platforms for custom packet processing, virtualized router platforms have been proposed [7]. To represent the data-flow oriented nature of packet processing and to provide flexible service composition, several application abstractions have been proposed (e.g., Click [8] for general router platforms, NP-Click [9] for network processor based routers).

The problem of mapping tasks to processing resources in network processing systems has been studied widely and several approaches have been proposed for static workloads [9], [10]. The Shangri-la project [11] developed a high level domain specific language compiler and runtime system and bears some similarities to our work, but focuses on pipeline structures rather than task distribution. Dynamic changes in workloads have been considered more recently in [12]–[14]. Our work differs from these prior approaches in that we use runtime profiling to obtain an understanding of the workload requirements. Also, we consider partitioned applications (instead of monolithic applications) in the task mapping process. This approach considers limitations in instruction store rather than the more unrealistic assumption that all packet processing functions can be performed on a single processor core (i.e., run-to-completion model) and arbitrarily parallelized. The idea of using task duplication has recently been developed in parallel [15]. Our work does not only consider processing requirement but also inter-processor communication to find mapping solutions.

III. Workload Mapping on Packet Processing System

Parallelism in multiprocessor packet processing systems can be exploited by processing multiple packets on different applications (or different instances of the same application) and by pipelining processing over partitioned applications. In our work, we define an "application" as a protocol processing step or a network service (e.g., IP forwarding, firewalling, VPN termination). Thus, there are multiple, different applications available on a router and different packets may traverse a different sequence of these applications as they are processed.

In this section, we first describe the representation of the packet processing applications as a task graph. Then, we formalize the problem that we are addressing in this work. The task graph can be annotated during runtime through profiling information. These annotations are used to determine which tasks to duplicate and how to map them to processing resources. Finally, dynamic adaptation to workload changes is considered.

A. Application Representation and Partitioning

The application representation should provide an architecture independent model of the workload that can be used with different mapping algorithms on different kinds of network processors.

1) Graph Representation: The granularity at which applications are represented determines the basic mapping unit that an algorithm works on. In general, the higher the level of representation is, the coarser mapping results are. Thus, it is important to consider at which level to represent such information. In the finest level of representation, individual processor instructions are considered. Such a representation requires a very large amount of information and is very complex to use in later stages of the mapping process (e.g., due to large overhead for profiling). In the coarsest level, the application is seen as a single monolithic block. Such a representation lacks an insight about the application and leads to trivial (and likely low performance) mapping results. We chose to represent applications at the level of "tasks," where tasks are basic functional blocks in the application similar to Click modules [8].

The workload of the system, consisting of a number of interdependent applications, is then represented as a graph. Applications are represented by nodes, and a directed edge indicates that there may exist some packets that require processing of the application from where the edge originates followed by the application to which the edge points. For simplicity, we assume that there is one node at which all packets enter the system and another node at which all packets leave the system.

An example of this graph structure is shown in Figure 1. There are several different applications (denoted by a_i) that represent packet processing steps. Arrows indicate the flow of packets through the system (and the resulting dependencies between applications). Since applications can be considered as separate entities on a packet processing system, inter-application parallelism is fully considered in this representation. In practice, inter-application parallelism is often too coarse for mapping and thus we also consider intra-application parallelism.

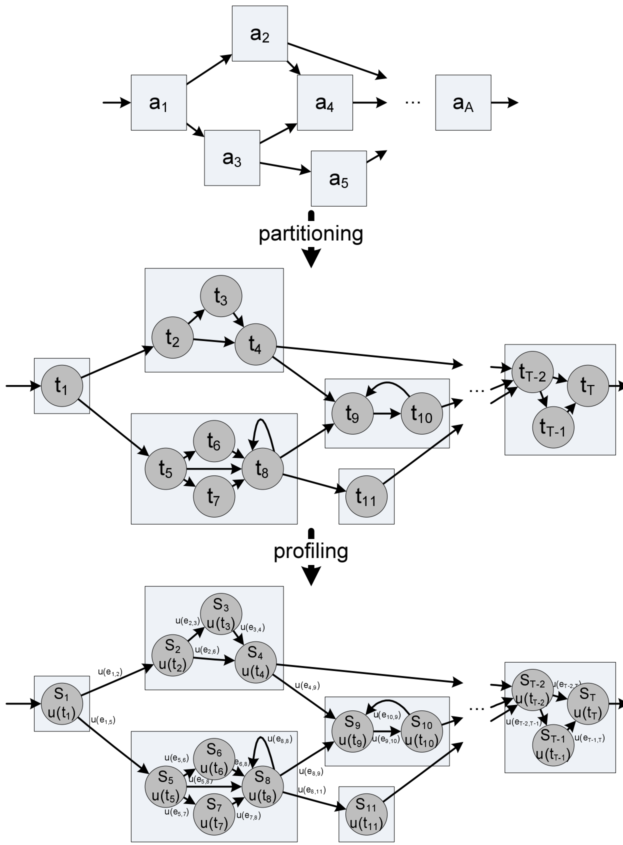


Fig. 1. Workload Partitioning and Profiling Steps.

2) Partitioning: To obtain a more detailed level of workload representation, we further divide each application into a subgraph with nodes representing tasks within the application. Each task represents a set of processing instructions and serves as the basic unit that is mapped to hardware processing unit. Semantically, these tasks represent fundamental processing operations that occur in the context of packet processing (e.g., protocol header extraction, loop within router lookup algorithm, checksum computation, etc.). Typically, such tasks can be determined by examining source code and identifying major functions or by starting from a Click representation of router functionality [8]. If different applications have some common functionality (i.e., extract a certain header field), they can share the corresponding tasks. This partitioning process is illustrated in Figure 1.

B. Problem Statement

Assume we are given the task graph of all subtasks in all applications by T task nodes $t_1, \ldots, t_T$ and directed edges $e_{i,j}$ that represent processing dependencies between tasks t_i and t_j. Also assume that we represent a packet processing system by N processors with M processing resources on each (i.e., each processor can accommodate M tasks and the entire system can accommodate $N \cdot M$ tasks). The goal of our work is to find a mapping M that assigns each of the T tasks to one of N processors: $m : \{t_1, \ldots, t_T\} \rightarrow [1, N]$. This mapping needs to consider the constraint of resource limitations: $\forall j, 1 \leq j \leq N : |\{t_i | m(t_i) = j\}| \leq M$.

In this context, there are several observations that need to be considered:

- Processing resources are typically hardware threads on a processor core. Most high-performance packet processing systems support hardware multi-threading and thus we consider a thread as a basic processing resource.
- Tasks may be mapped to multiple processing resources or none. If tasks are computationally demanding, a single processing resource may not be sufficient and thus multiple processing resources could be used. Since there are no inter-packet dependencies for most packet processing applications, such parallelization is easily possible. If a task is not used at all (e.g., rare exception processing), no processing resource may need to be allocated.
- The quality of the mapping can be measured by a number of different metrics (e.g., system utilization, power consumption, packet processing delay, etc.). In our work, we aim to find a mapping that provides the most balanced processor utilization. The reasoning behind this metric is that such a mapping can provide the highest overall throughput without overloading any particular processor core.

When considering the quality of a mapping, the mapping problem becomes an optimization problem for the chosen metric. For example, when considering system utilization, the goal is to find a mapping such that the difference between the maximum and minimum processor utilization across the system is minimized. Clearly, it is impossible to perform this optimization without more detailed knowledge of the processing demands of each task and the paths that packets take through the system. To obtain this information, we use runtime profiling.

C. Runtime Profiling

The workload of the packet processing system is affected by two factors: first, the computational characteristics of all tasks in the system; second, the network traffic that exercises the processing system. In order to derive an optimal mapping, both need to be quantified and considered in the mapping process.

Many systems have used offline profiling information to obtain processing characteristics of tasks. However, these offline solutions cannot consider variation in application sequences that are due to changes in network traffic that occur during runtime. Also, processing requirements may be data-dependent and thus change depending on packet data (which cannot be predicted). Therefore, we use a runtime profiling approach, where profiling information is collected while the system is operational.

We collect the following information:

- Task Service Time s_i: For each task t_i, we determine the service time s_i (measured, for example, in number of instructions executed per packet). Since this value may be different for each packet, we consider s_i as a sample

from a random variable S_i. We assume the distribution of S_i matches the empirical observations of s_i.

- Edge Utilization $u(e_{i,j})$: At the completion of each processing tasks, we observe where the packet is processed next. This transition from task t_i to task t_j is denoted as utilization $u(e_{i,j})$ of edge $e_{i,j}$.
- Task Utilization $u(t_i)$: Based on edge utilization (or through direct profiling), we can derive the utilization of a particular task t_i, which is denoted by $u(t_i)$.

Using this information, we can annotate the workload graph with execution time distributions S_i for each task. This is illustrated in Figure 1. Since edge utilization (and thus task utilization) changes over time, we denote them as dependent on time parameter τ: $u^\tau(t_i)$ and $u^\tau(e_{i,j})$. This time-dependence is further considered in Section III-F, where dynamic adaptation is discussed. We assume that the service time distribution is not time-dependent (although that could be considered in a straightforward extension of this work).

D. Task Duplication

One of the main goals of task mapping is to fully utilize the available system resources and thus support the highest possible data rate. When considering the load that a task places on a processing resource, we need to consider not only how computationally demanding a task is (i.e., expected service time $E[S_i]$), but also how frequently it is used (i.e., task utilization $u(t_i)$). Thus, we define w_i as the amount of "work" that is imposed by task t_i:

$$w_i = u(t_i) \cdot E[S_i]. \quad (1)$$

Clearly, the amount of work for different tasks may vary significantly. Note that this imbalance is not only due to differences in task size when partitioning, but also due to differences in utilization. The latter is dependent on dynamic traffic requirement. Therefore the balance issue cannot be addressed by better partitioning algorithms. Instead, it is necessary to dynamically balance the work for each task.

1) Duplication Process: Since we cannot change the service time for a task, we adapt the work for a task by changing its utilization. We accomplish this goal by duplicating a task. This task duplication creates an additional instance of a task that is fully connected to the same predecessor and successor tasks as the original task. We assume that the predecessor distributes packets uniformly among all task instances and thus effectively reduces the edge utilizations leading to each task instance. This process is illustrated in Figure 2.

We use parameter d_i to indicate the number of duplicated instances that exist for task t_i. These instances are named $t_i^1, t_i^2, \ldots, t_i^{d_i}$. Any incoming edge $e_{j,i}$ from tasks t_j to t_i is duplicated: $e_{j,i^1}, e_{j,i^2}, \ldots, e_{j,i^{d_i}}$. Similarly, outgoing edges are duplicated. Due to the reduced edge utilization of $u(e_{j,i})/d_i$, fewer packets are processed by each task instance and the task utilization decreases to $u(t_i)/d_i$. Correspondingly, the amount of work required by each task instance is denoted as w_i':

$$w_i' = \frac{u(t_i)}{d_i} \cdot E[S_i]. \quad (2)$$

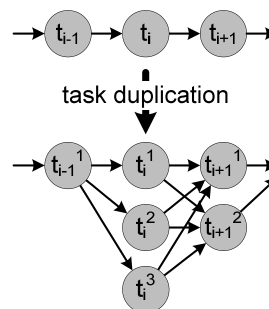


Fig. 2. Task Duplication Example with d_{i-1}=1, d_i=3, and d_{i+1}=2.

2) Duplication Choice: The main question remaining is: How to determine the best set of d_i (i.e., which task to duplicate how many times)? Our goal is to balance the amount of work that each task performs in order to simplify the mapping process. Thus, the ideal scenario would be one where $w_1' = w_2' = \cdots = w_T'$. However, such a scenario may require very large values for d_i if w_i' do not share common factors. Such a solution would conflict with the constraint that we cannot have more tasks instances than processing resources (i.e., $\sum_i^T d_i \leq N \cdot M$).

To make duplication choices while observing this processing resource constraint, we use a greedy approach shown as Algorithm 1. While processing resources are available, we identify the task that has the highest w_i' value. Adding a duplicated task instance to this task reduces the amount of work done by each instance because $\frac{u(t_i)}{d_i+1} \cdot E[S_i] < \frac{u(t_i)}{d_i} \cdot E[S_i]$ for any d_i, $u(t_i)$, and $E[S_i]$. We repeat this process until all processing resources are used.

Algorithm 1 Task Duplication Algorithm.

```
1: while Σ_{i=1}^T d_i < N · M do
2:     j ← argmax_i w'_i
3:     d_j ← d_j + 1
4: end while
```

Depending on the number of tasks in the workload and the number of available processing resources, there may be more tasks than resources (i.e., $T > N \cdot M$). In such a case, either the partitioning needs to be repeated to reduce T, or multiple tasks need to be combined onto a single processing resource. While both approaches are possible, we do not consider them further in this paper and assume $T \leq N \cdot M$.

E. Task Mapping Algorithm

Given a workload graph with duplicated task instances, we need to map each task to a packet processing resource. This process is illustrated in Figure 3. One of the main concerns in this context is where tasks that are dependent on each other are mapped relative to each other. If a task passes packets to another task and both tasks are placed on the same processor, then state can efficiently be transferred through local registers. If the tasks reside on different processors, the processor interconnect needs to be used for the transfer.

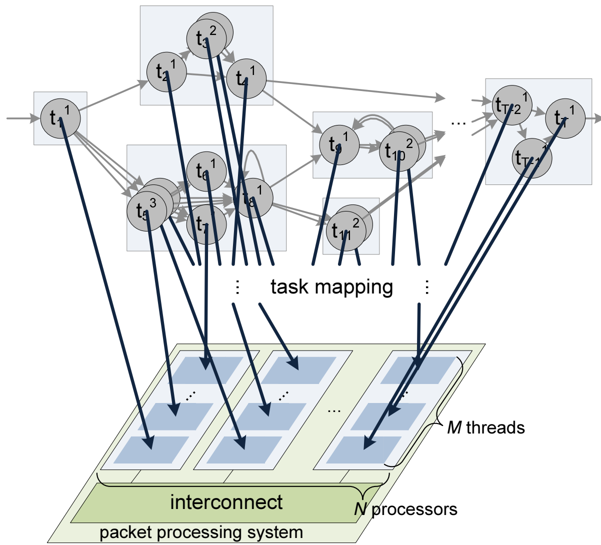


Fig. 3. Task Mapping Step.

Algorithm 2 UDFS Task Mapping Algorithm.

```
1: function map_next(i,p)
2: while ∃ e_{i,j} with t_j unmapped do
3:     k ← argmax_j(u(e_{i,j}))
4:     if tasks_allocated_to(p) ≤ M then
5:         m(t_k) ← p
6:         p ← map_next(k,p)
7:     else
8:         m(t_k) ← p + 1
9:         p ← map_next(k,p + 1)
10:    end if
11: end while
12: return p
13:
14: function map()
15: m(t_1) ← 1
16: map_next(1,1)
17: return m
```

Therefore, an effective mapping algorithm needs to consider two important aspects:

- Task Locality: Tasks t_i and t_j that are connected through an edge e_{ij} (or through a short path of edges), in practice often may share data structures. Thus, placing these tasks on the same packet processing engine may improve the efficiency of the system (e.g., caching is more effective, locks on data structures cause less overhead, etc.).
- Interconnect Usage: An edge e_{ij} between two tasks t_i and t_j that are not located on the same packet processor (i.e., $m(t_i) \neq m(t_j)$) implies that packets need to be transferred across the processor interconnect. The utilization $u(e_{ij})$ quantifies the amount of interconnect bandwidth that is necessary for this communication. Ideally, the interconnect usage should be kept to a minimum to avoid queuing and processing backlog.

In light of these goals, we use the utilization-based depth-first (UDFS) algorithm shown as Algorithm 2 for task mapping. The algorithm greedily clusters tasks on a processor until all processing resources are fully utilized. The key aspect of the algorithm is the order in which the task graph is traversed. High-utilization edges are traversed first to increase task locality and reduce interconnect usage.

A more detailed description of UDFS is as follows: We initially map node t_1, which is assumed to be the ingress node for all traffic, to the first processor. Then, using the *map_next* function, we search among all outgoing edges to find that with the highest utilization. If there are still resources available on the same processor, the task that is pointed to by this edge is mapped to the same processor. Otherwise it is mapped to the next processor. This process is repeated recursively to achieve depth-first mapping. The recursion terminates when a node has no outgoing edges to unmapped tasks (e.g., egress node). The variable p keeps track of which processor is currently being used for task allocation.

The following observations related to the algorithm need to be considered: (1) The algorithm maps tasks and their duplicates. To simplify notation, only tasks are mentioned. (2) If the ingress task is different from t_1, the algorithm can be easily adapted. (3) We assume that a packet transfer between processors is the basic unit of interconnect usage. In some cases, it may be possible that the interconnect usage is variable. This can occur when different amounts of processing state needs to be sent between processors. In such a scenario, line 3 of the algorithm would use a different function inside the argmax function.

F. Dynamic Adaptation

Dynamic adaptation of task mapping is crucial for packet processing systems. The processing workload required by network traffic cannot be known in advance since end-systems may send packets to any arbitrary destination using any protocol. Thus, a packet processing system needs to either (1) over-provision for any possible traffic scenario or (2) dynamically adapt. With an increasing diversity of services that are provided in packet processing systems, the first choice is becoming less feasible. Thus, we need to consider how task mapping can be adapted to changes in traffic.

Conveniently, run-time profiling monitors dynamic trends in the processing workload. This information is reflected in utilization parameters $u^\tau(t_i)$ and $u^\tau(e_{i,j})$. Thus, it can be directly used in the mapping process. An important question is how frequently to update this information and how frequently to revise the task mapping. The utilization information should be collected over a reasonably large number of packets to average out short packet bursts that are not representative of the overall workload. The interval between task mappings should depend on how much the workload changes. In related work, different mechanisms have been proposed (e.g., mapping based on length of inter-processor queues [12], mapping based on fixed intervals optimized for workload [13]).

IV. Experimental Results

We evaluate the effectiveness of our workload profiling and task mapping process through simulation and analysis.

A. Experimental Setup

Our experimental setup follows the methodology described above and can be divided into two phases:

1) Workload Profiling Phase: During workload profiling, we use PacketBench [16] to evaluate the processing requirements of each packet in a trace of network traffic. PacketBench provides an instruction trace of each processor instruction executed and thus allows us to accurately determine utilization parameters $u^{\tau}(t_i)$ and $u^{\tau}(e_{i,j})$ for each interval τ and the distribution of service time S_i (measured in instructions executed).
2) Task Mapping Phase: During task mapping, tasks are duplicated and mapped as described above. This process is repeated for each interval τ. We use analytical methods to evaluate mapping results.

For our experiments, we assume a packet processing system with $N = 8$ processors with $M = 8$ threads each (similar to the Intel IXP2400 network processor). The processor interconnect provides connectivity from any processor to any other processor. Memory accesses and contention on memory interfaces is not considered within the scope of this paper. We assume mapping takes place at intervals of 1000 packets.

We use two different packet traces in our experiments in order to exercise the system with network traffic that exhibits different levels of workload dynamics:

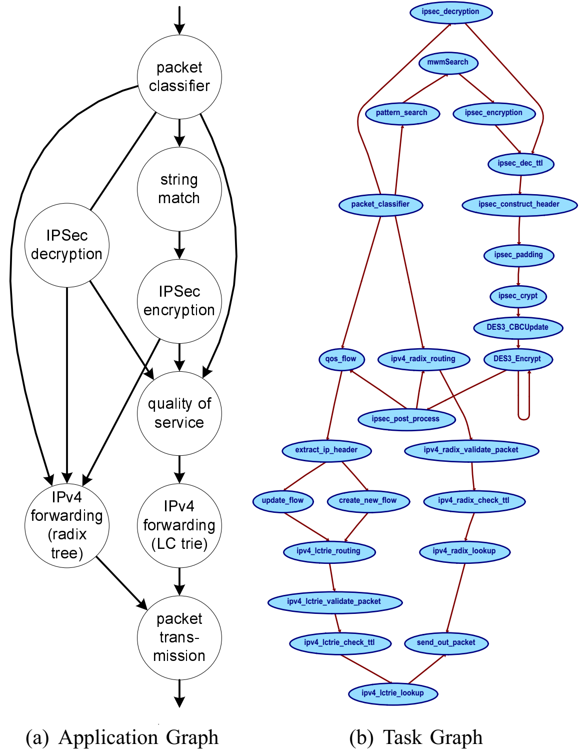
- Trace 1: This trace is obtained from the Internet uplink of our institutional network. It represents real network traffic and exhibits a low amount of dynamic variation. The trace is 100 intervals long.
- Trace 2: This trace was generated synthetically by splicing several different traces together. The resulting workload changes dramatically every 10 intervals to require a drastic change in allocated processing tasks. The trace is 40 intervals long.

The processing applications in our workload are shown in Figure 4(a) with their respective dependencies. By partitioning these eight applications, we obtain the task graph shown in Figure 4(b). The 25 tasks shown in Figure 4(b) are labeled with their functional descriptions. Edges illustrate the possible paths of packets through the system.

B. Profiling

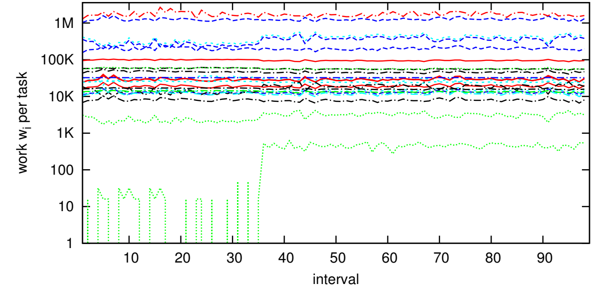
The results of the profiling phase are shown in Figure 5. For each processing task from Figure 4(b), we show the amount of processing work, w_i, that is necessary. Recall that this value depends on the processing complexity of the task and its utilization (Equation 1).

First, we observe that there is a very large difference between tasks in terms of processing requirements (note the logarithmic scale on the y-axis). The variation of w_i for any given task is low for Trace 1 (Figure 5(a)). In contrast,

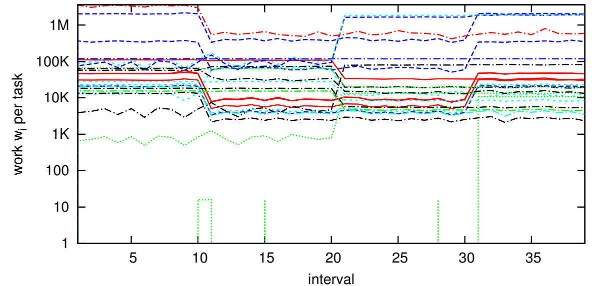


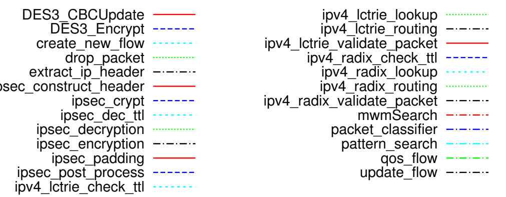
(a) Application Graph (b) Task Graph

Fig. 4. Experimental Workload.



(a) Trace 1





(b) Trace 2

Fig. 5. Work w_i of Tasks t_i over Time. Legend applies to both figures.

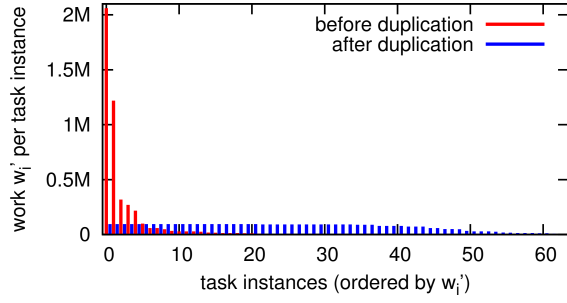


Fig. 6. Distribution of Work w_i' per Task Instance Before and After Duplication. Data is from one single interval of Trace 1. Tasks are ordered by decreasing w_i'.

Figure 5(b) shows high variations due to the changes in network traffic every 10 intervals.

These profiling results provide evidence for two observations we have made earlier: (1) there is a big difference in processing requirements among tasks and (2) these requirements change dynamically as network traffic changes.

C. Duplication

Due to the large differences in work w_i, we use duplication of selected tasks to obtain a more balanced workload. The resulting work w_i' (from Equation 2) is shown in Figure 6. This figure shows the amount of work per task instance before and after duplication for one interval from Trace 1. Before duplication, only 25 task instances exist and their processing requirements differ by several orders of magnitude. After duplication, we have 64 task instances (since $N \cdot M = 64$ in our experimental setup) with very balanced w_i (except for the smallest tasks). These data illustrate how difficult it would be to find a balanced mapping when using tasks without duplication. A single task with large processing requirements would represent a bottleneck in the packet processing system.

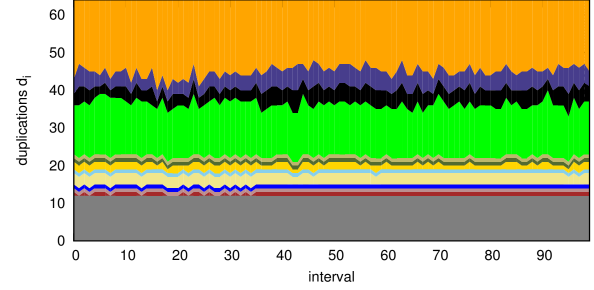
To illustrate the duplication process over time, Figure 7 shows the number of duplications d_i for each task t_i. The data are plotted cumulatively, thus there is always a total of 64 task instances. To improve readability of the figure, all tasks with only a single instance $d_i = 1$ are aggregated into "other." For both traces, it can be observed that some tasks may not have any instance ($d_i = 0$) for some intervals (e.g., *ipsec_decryption*). Also, the number of instances changes with the requirement of network traffic as can be seen clearly in Figure 7(b).

D. Mapping

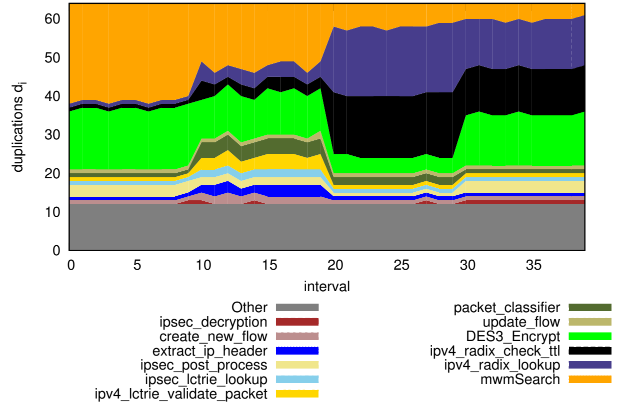
To evaluate the quality of the mapping algorithm, we consider two metrics:

- Average Processor Utilization $\overline{u}$: The average utilization $\overline{u}$ of all processors is the sum of all work allocated to each processor divided by N times the maximum allocation:

$$\overline{u} = \frac{\sum_{j=1}^{N} \left(\sum_{\{i|m(t_i)=j\}} w_i' \right)}{N \cdot \max_j \left(\sum_{\{i|m(t_i)=j\}} w_i' \right)}. \quad (3)$$



(a) Trace 1



(b) Trace 2

Fig. 7. Cumulative Duplications d_i for all Tasks t_i over Time. Legend applies to both figures. Tasks with only a single instance ($d_i = 1$) are aggregated into "other."

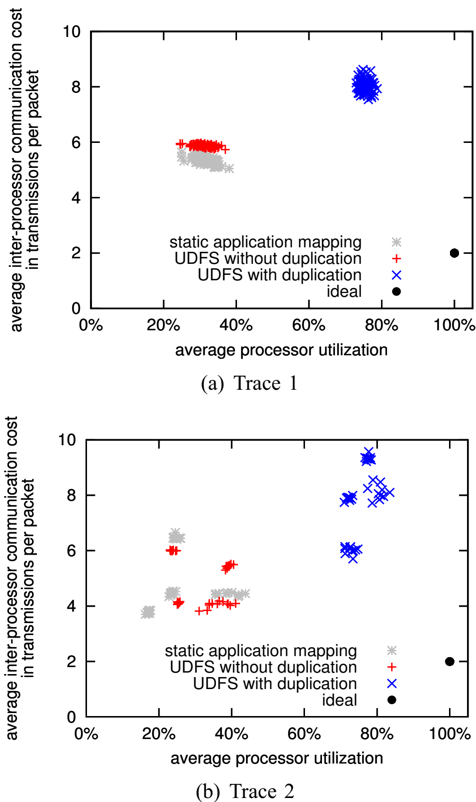
When each processor's work allocation is close to the maximum, then the overall average utilization is high. Higher utilization implies that more work gets done and more packets get processed (since the total amount of work $\sum_{i=1}^{T} w_i$ is constant for any mapping result). Thus, utilization is directly related to the maximum line rate (i.e., throughput) R of the packet processing system: $R \sim \overline{u}$. Thus, higher utilization $\overline{u}$ indicates higher system performance.

- Average Inter-Processor Communication Cost $\overline{c}$: The average communication cost $\overline{c}$ represent the number of times a packet has to be sent across the processor interconnect:

$$\overline{c} = \sum_{\{i,j|m(t_i) \neq m(t_j)\}} u(e_{ij}). \quad (4)$$

At a minimum, each packet has to be sent once from the incoming interface to a processor and once from the processor to the outgoing interface. Thus, $\overline{c} \geq 2$. Higher values for $\overline{c}$ imply more load on the interconnect. Therefore, lower values of $\overline{c}$ are desirable.

Figure 8 shows a comparison of the performance of three different algorithms using metrics $\overline{u}$ and $\overline{c}$. As baseline, we show *static application mapping*, which represents the conventional approach to task management on multicore packet processing systems. Each application a_i is allocated to a



(a) Trace 1

(b) Trace 2

Fig. 8. Interconnect Bandwidth $\overline{c}$ in Comparison to Processor Utilization $\overline{u}$ for Different Mapping Algorithms.

TABLE I
COMPARISON OF UDSF MAPPING TO STATIC APPLICATION MAPPING.

	Communication Cost $\overline{c}$	Throughput R
Trace 1	1.49×	2.39×
Trace 2	1.64×	2.89×

different processor. The UDFS algorithm is shown in two instances – without duplication and with duplication. The latter is the task mapping approach that we propose in our paper. The prior is an intermediate result to illustrate the importance of task duplication. The ideal scenario of full utilization and a two packet transmission (one ingress, one egress) is also shown for comparison. The data in Figure 8 show clearly that UDFS mapping with task duplication achieves by far the highest system utilization $\overline{u}$ and thus the highest data rate R. UDFS mapping without task duplication is practically equivalent to static mapping since the imbalance in the amount of work w_i per task prevents an effective utilization processors.

The overall performance improvement of UDFS (with duplication) over conventional static application mapping is shown in Table I. An increase in throughput (due to $R \sim \overline{u}$) of 2.39–2.89× can be achieved at a cost of 1.49–1.64× higher inter-processor communication.

V. SUMMARY AND CONCLUSIONS

We have presented a methodology for profiling packet processing workloads in order to effectively distribute tasks onto processing resources. For multicore systems such runtime management is crucial in order to adapt to changing network workloads and provide high system performance. Our UDFS algorithm with task duplication can provide a more than twofold speedup over conventional task mapping at a cost of around 50% higher inter-processor communication. For practical systems, this presents a significant improvement over the current state of the art.

REFERENCES

[1] A. Feldmann, "Internet clean-slate design: what and why?" *SIGCOMM Computer Communication Review*, vol. 37, no. 3, pp. 59–64, July 2007.

[2] W. Eatherton, "The push of network processing to the top of the pyramid," in *Keynote Presentation at ACM/IEEE Symposium on Architectures for Networking and Communication Systems (ANCS)*, Princeton, NJ, Oct. 2005.

[3] T. Wolf, "Service-centric end-to-end abstractions in next-generation networks," in *Proc. of Fifteenth IEEE International Conference on Computer Communications and Networks (ICCCN)*, Arlington, VA, Oct. 2006, pp. 79–86.

[4] K. L. Calvert, J. Griffioen, and S. Wen, "Lightweight network support for scalable end-to-end services," in *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, Pittsburgh, PA, Aug. 2002, pp. 265–278.

[5] T. Wolf, "Challenges and applications for network-processor-based programmable routers," in *Proc. of IEEE Sarnoff Symposium*, Princeton, NJ, Mar. 2006.

[6] J. S. Turner, P. Crowley, J. DeHart, A. Freestone, B. Heller, F. Kuhns, S. Kumar, J. Lockwood, J. Lu, M. Wilson, C. Wiseman, and D. Zar, "Supercharging PlanetLab: a high performance, multi-application, overlay network platform," in *SIGCOMM '07: Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, Kyoto, Japan, Aug. 2007, pp. 85–96.

[7] T. Anderson, L. Peterson, S. Shenker, and J. Turner, "Overcoming the Internet impasse through virtualization," *Computer*, vol. 38, no. 4, pp. 34–41, Apr. 2005.

[8] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click modular router," *ACM Transactions on Computer Systems*, vol. 18, no. 3, pp. 263–297, Aug. 2000.

[9] N. Shah, W. Plishker, and K. Keutzer, "NP-Click: A programming model for the Intel IXP1200," in *Proc. of Second Network Processor Workshop (NP-2) in conjunction with Ninth IEEE International Symposium on High Performance Computer Architecture (HPCA-9)*, Anaheim, CA, Feb. 2003, pp. 100–111.

[10] S. D. Goglin, D. Hooper, A. Kumar, and R. Yavatkar, "Advanced software framework, tools, and languages for the IXP family," *Intel Technology Journal*, vol. 7, no. 4, pp. 64–76, Nov. 2003.

[11] 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," in *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, Chicago, IL, June 2005, pp. 224–236.

[12] R. Kokku, T. Riché, A. Kunze, J. Mudigonda, J. Jason, and H. Vin, "A case for run-time adaptation in packet processing systems," in *Proc. of the 2nd Workshop on Hot Topics in Networks (HOTNETS-II)*, Cambridge, MA, Nov. 2003.

[13] T. Wolf, N. Weng, and C.-H. Tai, "Run-time support for multi-core packet processing systems," *IEEE Network*, vol. 21, no. 4, pp. 29–37, July 2007.

[14] X. Huang and T. Wolf, "A methodology for evaluating runtime support in network processors," in *Proc. of ACM/IEEE Symposium on Architectures for Networking and Communication Systems (ANCS)*, San Jose, CA, Dec. 2006, pp. 113–122.

[15] A. Mallik and G. Memik, "Automated task distribution in multicore network processors using statistical analysis," in *Proc. of ACM/IEEE Symposium on Architectures for Networking and Communication Systems (ANCS)*, Orlando, FL, Dec. 2007, pp. 67–76.

[16] R. Ramaswamy and T. Wolf, "PacketBench: A tool for workload characterization of network processing," in *Proc. of IEEE 6th Annual Workshop on Workload Characterization (WWC-6)*, Austin, TX, Oct. 2003, pp. 42–50.