Supplementary Material: Runtime Task Allocation in Multi-Core Packet Processing Systems

Qiang Wu, *Student Member, IEEE,* Tilman Wolf, *Senior Member, IEEE*

This supplementary material provides additional discussion, algorithms, and results that go beyond what is presented in the article.

I. Workload Representation

Section III.A of the article discusses workload representations. The following three representations of workload move progressively from a user-friendly representation to a workload model that can be efficiently implemented:

- Data Path Specification: From a user's point of view, packet processing systems of routers implement several network, transport, or application layer functions. These packet processing "applications" range from protocol processing steps (e.g., IP forwarding, firewalling, VPN termination) to more complex network services [1], [3]. The data path specification determines which sequences of applications can be traversed by packets. Typically, there are multiple, different applications available on a router, and different packets traverse a different sequence of these applications as they are being processed. Generally, the application representation – while easy to understand for a user – does not provide sufficient detail to allow for a high-performance implementation.
- Graph Representation: To provide more details about the operation of the data path, the graph representation uses nodes to represent processing tasks and directed edges to denote communication paths (i.e., dependencies). Each task denotes a schedulable unit that runs on the hardware platform and performs a set of operations on packets it receives. This representation is used in our system to collect profiling information and to run the task mapping algorithm.
- Implementation with Click Elements: The Click modular router is a software system that allows the construction of custom data path functions from basic function blocks called "elements" [2]. Each element belongs to a corresponding C++ class, which implements a specific functionality in the packet processing procedure. Communication between Click elements is performed via connections and queues. Each element has input and output ports, which serve as the endpoints of packet communication between elements. Connections between elements in form of directed edges represent a potential path on which packets can travel.

Q. Wu is with Juniper Networks, Inc.; T. Wolf is with the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA, 01003, USA; email: qwu@juniper.net, wolf@ecs.umass.edu

II. Task Mapping

This section presents the mapping algorithm that is used in our work after task duplication has been performed.

Mapping tasks to processing resources based on their processing requirements is a load balancing problem. As discussed in the article, the packing problem of fitting processing-intensive tasks with less processing-intensive tasks onto a shared processing resource is intractable. However, the use of task duplication allows us to control the work requirement of a task as defined by Equation (1) in the article by adjusting d_i as stated in Equation (2) in the article. (Note that this flexibility is not available for the general task mapping problem, but can be exploited in the network processing domain.) Given a set of balanced tasks, it is much simpler to determine a balanced mapping as we discuss below.

A. 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. For each task, t_i, its utilization $u(t_i)$ and its service time S_i is given. 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 (1) determine the optimal number of duplicates d_i for each task t_i and (2) find a mapping m that assigns each of the T tasks (and their duplicates) to one of N processors: $m : \{t_1, \ldots, t_T\} \rightarrow [1, N]$. This mapping needs to consider the constraint on resource limitations:

$$\forall j, 1 \leq j \leq N : |\{t_i | m(t_i) = j\}| \leq M. \quad (1)$$

The quality of the mapping can be measured by several 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 (i.e., a mapping such that the difference between the maximum and minimum processor utilization across the system is minimized) and effectively exploits locality in processing. Task duplication is the basis for load balancing and mapping is used to achieve locality. 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. Thus, runtime profiling information is essential.

B. Task Mapping Algorithm

Given a workload graph with duplicated task instances, we need to map each task to a packet processing resource. 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.). If there is a choice of tasks, the task t_j with higher utilization $u(t_j)$ is preferred as more packets traverse between the current task t_i and t_j. Thus, placing both tasks on the same processor increases locality for more packets.
- Workload Balance: The resulting mapping should balance the total work allocated to all processors. Fortunately, since all task instances require approximately the same amount of work (due to the previous duplication step), the algorithm can derive a balanced solution simply by allocating the same number of tasks to each processor. It is not necessary to solve a complex packing problem to balance the amount of work on each process.

In light of these goals, we use the utilization-based depth-first (UDFS) algorithm shown as Algorithm 1 for task mapping. The algorithm greedily clusters tasks on a processor until all processing resources are fully utilized. The order of graph traversal determines the allocation of tasks to processor cores. High-utilization downstream tasks are traversed first to increase task locality.

Algorithm 1 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(t_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
```

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 the highest utilized downstream task. 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. Note that the algorithm maps tasks and their duplicates, but to simplify notation, only tasks are mentioned.

An alternative to the depth-first search in UDFS is a breadth-first search. In that case, task allocation roughly follows a software-pipelining approach (rather than the run-to-completion approach in UDFS). In scenarios with multiple different processing paths for network traffic, UDFS is preferable due to higher locality and fewer packet transitions between processors.

In this paper, we focus on the processing aspect of a router's data path operation. Intercommunication cost between tasks is closely related to the underlying hardware architecture. For example, in SMP systems with a shared memory bus, intercommunication between two tasks can be implemented either via cache (when tasks are on a same core) or memory (when tasks are on different cores). On systems with dedicated inter-processor communication channel (e.g. next-neighbor register on Intel IXP 2400), intercommunication cost can be largely reduced by utilizing such channels. The proposed UDFS algorithm tries to map neighboring tasks that have most intercommunication on same core. Therefore it is applicable to most systems. Detailed analysis on such cost in the UDFS algorithm can be found in [4].

III. Evaluation Setup

Section V.A of the article discusses the prototype system setup consisting of Systems I–IV. This section provides additional discussion on their configuration, including the full task graph.

Systems III and IV are designed based on the concept of task graph that is inherent to Click. Figure 1 shows the data path configuration. For each task, we use multiple Click elements to make sure it is schedulable across all processor cores. For example, the inset in Figure 1 illustrates the task `ipsec_postproc_b`, which contains a Click element `Unqueue` that can be executed on any data path processor core. This operation pulls packets from the task's input queue and the processes them all the way to the end of the task. Between tasks, we use `MSQueue` elements as edges. On systems only permitting data exchange in shared memory, duplicating edges has very limited benefits. We therefore use a large queue for each edge, and map all logically duplicated edge instances to their original queue.

Since inter-task queues do not get duplicated during task duplication, it is possible that multiple element instances pull packets from a single queue. Therefore, it is necessary to modify the `MSQueue` element to allow multiple pulls. We also implement a blocking mechanism for `MSQueue` that allows it to stall an upstream tasks' processing in case the queue is full.

We also extended each Click kernel thread with a profiling database, which records utilization of each task as well as its average service time using the processors' high-resolution

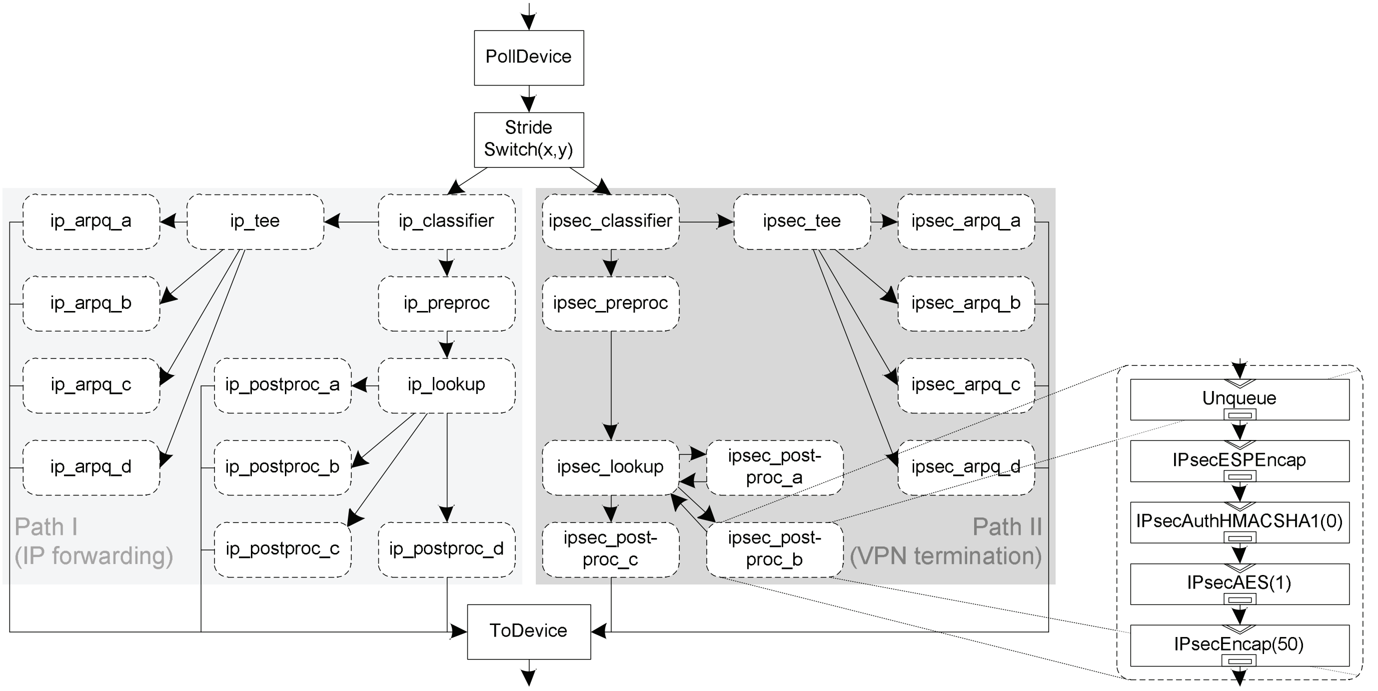

Fig. 1. Tasks in Click Router Configuration Used in Evaluation. Tasks contain multiple Click elements (as shown for `ipsec_postproc_b`).

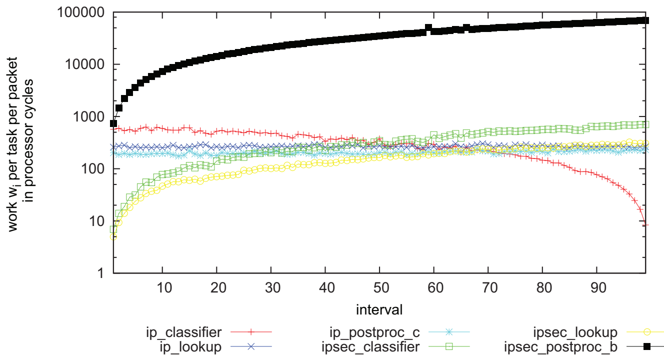

Fig. 2. Work w_i of Six Tasks over Different Intervals.

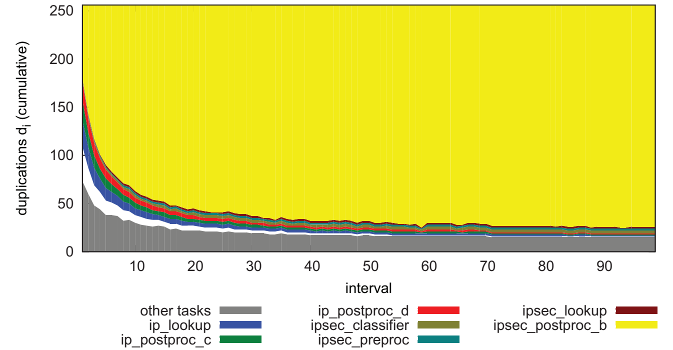

Fig. 3. Cumulative Duplications for Tasks over Different Intervals.

time stamp counter (TSC) register. A user space program reads profiling information periodically from each kernel thread, calculates duplication and mapping results for the task graph, and generates a (potentially different) Click router configuration that is then installed in the prototype system. In our prototype, the runtime system updates the configuration after fixed intervals.

IV. Evaluation

This section provides additional evaluation and analysis results.

A. Profiling

To illustrate the functionality of the profiling component in our runtime system, we show in Figure 2 the profiling information for six characteristic schedulable elements over the range of all processing workloads (Figure (5) in the article). The y-axis of this figure shows the total work per task per packet (as defined in Equation (1) in the article) in processor cycles. Initially, the `ip_classifier` and `ip_lookup` tasks require as much work as `ipsec_postproc_b`. As IPSec traffic increases, `ipsec_postproc_b` dominates processing requirements. The other tasks in the system follow similar patterns.

B. Task Duplication and Mapping

The profiling information from Figure 2 is used to duplicate tasks. Figure 3 shows the number of times each of the tasks is duplicated. For simplicity, several tasks are combined into "other tasks." The total number of available schedulable threads for each of the $N = 4$ processor cores is assumed to be $M = 64$. Thus, a total of 256 tasks are created. As expected, the tasks that require most processing are duplicated most frequently (initially, `ip_classifier` and `ip_preproc`, and later `ipsec_postproc_b`).

TABLE I
COMPUTATIONAL AND SPACE COMPLEXITY OF ALGORITHMS.

Algorithm	Computational complexity	Space complexity
Task duplication	$\mathcal{O}(MN \log T)$	$\mathcal{O}(T)$
Task mapping	$\mathcal{O}(MN + E)$	$\mathcal{O}(MN + E)$

C. Runtime Adaptation Algorithm Complexity

There are two main algorithms in the runtime adaptation system: Task duplication and task mapping. The computational complexity and space complexity of each algorithm is shown in Table I. The task duplication algorithm requires the sorting of tasks once and then a sorted insert for each duplicated task. Since we assume $T \leq M \cdot N$, the computation cost is bounded by $\mathcal{O}(MN \log T)$. (Note that the Algorithm shown as Algorithm 2 in the article does not use an optimized sorted insert to keep the description simple. Thus, this implementation of the algorithm has a running time proportional to MNT.) The space complexity of the duplication algorithm is bounded by T since one value of d_i needs to be stored for each task. The mapping algorithm has a computational requirement of $\mathcal{O}(MN + E)$, where E is the number of edges in the task graph. Each of the $M \cdot N$ task instances need to be mapped and each of the E may need to be traversed (even just to determine that the task instance that is pointed to by the edge is already mapped). The space requirement is $\mathcal{O}(MN + E)$ since each of the task instances and each of the edges need to be stored.

REFERENCES

[1] 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.

[2] 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.

[3] T. Wolf, "In-network services for customization in next-generation networks," *IEEE Network*, vol. 24, no. 4, pp. 6–12, Jul. 2010.

[4] Q. Wu and T. Wolf, "Dynamic workload profiling and task allocation in packet processing systems," in *Proc. of IEEE Workshop on High Performance Switching and Routing (HPSR)*, Shanghai, China, May 2008.