Open access • Journal Article • DOI:10.1109/TPDS.2012.56

Runtime Task Allocation in Multicore Packet Processing Systems — Source link

Qiang Wu, Tilman Wolf

Institutions: Juniper Networks, University of Massachusetts Amherst

Published on: 01 Oct 2012 - IEEE Transactions on Parallel and Distributed Systems (IEEE)

Topics: Packet processing, Multi-core processor, Network processor, Scheduling (computing) and Forwarding plane

Runtime Task Allocation in Multi-Core Packet Processing Systems

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

***Abstract*—Computer networks require increasingly complex packet processing functions in the data plane to adapt to new requirements. To meet performance demands, packet processing systems on routers employ multiple processor cores. To efficiently utilize processing resources in such systems, we propose a novel methodology for allocating tasks to processors. The main idea is to obtain runtime profiling information and to duplicate tasks with heavy processing requirements. Using our duplication algorithm, a balanced workload can be obtained and the complexity of packing tasks with different processing requirements can be reduced. By translating traffic characteristics into processing requirements, the system is able to adapt to dynamic changes in the workload and balance the utilization of all processing resources to maximize system throughput. Our approach can adapt to any traffic change in a single iteration, whereas existing adaptive approaches may require multiple steps. Results from our prototype implementation based on the Click modular router show that our system only requires on average 5.3%–31.5% of the adaptation steps that are necessary in iterative systems. In addition, our system achieves a throughput that is 1.32 times higher than the throughput achieved with symmetric multiprocessing support with general-purpose task allocation.**

***Index Terms*—network router, multi-core processor, network processor, task allocation, scheduling.**

I. Introduction

The complexity of operations performed in the data plane of today's Internet has expanded significantly beyond the simple store-and-forward concepts proposed in the original architecture [5]. Packet processing steps in IP routers are numerous and include packet classification, content inspection, traffic shaping, and accounting [6]. It can be expected that the trend toward more functionality and complexity in the data plane continues in order to accommodate demands for more security features, operational controls, and new data path services. A similar need for complex data path functionality can be observed in the proposed architectures for the next-generation Internet [7], where virtualized router platforms need to perform packet processing operations for several parallel networks with different data path functionality.

This need for flexible packet processing systems has led to development of router architectures with programmable packet processors that handle packets in the data path. These systems employ highly parallel, programmable packet processors ranging from embedded multi-core systems-on-a-chip (i.e., network processors) to programmable logic devices with high-performance I/O capabilities. However, multi-core programming remains a challenge on parallel packet processing systems as complexity of network services increases [2]. Increasing diversity and complexity of networking workloads makes the traditional run-to-completion programming model, which implements the full functionality of the router on each core, unsuitable for these systems. To exploit the parallelism in multi-core systems, programming models have been developed to break monolithic network processing applications into smaller tasks that can be distributed across multiple cores [3], [12], [20]. A key problem in this context is to determine an allocation of tasks to processing engines that maximizes system throughput (or meets another optimization goal).

Task allocation in multi-core packet processing systems is challenging due to several reasons. First, processing engines, memories, and the I/O system of typical network processors are tightly coupled. Overloading one system component may present a bottleneck that impacts the performance of the overall system. Therefore, it is important that the task allocation process considers the load placed on each component. Second, network traffic changes dynamically and with it the demand for different processing functions (e.g., IPSec processing varies with amount of VPN traffic). This variability requires packet processing systems to adapt to changes in processing demands for different tasks. Thus, it is necessary to determine task allocation *at runtime* rather than through offline optimization.

The task allocation process presented in this paper can achieve a near optimal allocation of tasks and can adapt to changes in traffic and processing workloads. The main idea of our approach is to obtain runtime profiling information that is used to understand the traffic and processing characteristics. Based on this information, we determine how many parallel instances of each task are necessary to avoid bottlenecks. This step also ensures that processing tasks incur nearly equal amounts of work and thus can be allocated more easily without having to solve a complex packing problem. Through continued profiling and updating of these steps, runtime adaptation is achieved. A particularly important aspect of our process is its simplicity that ensures that it is feasible to implement such a runtime system on resource-constrained packet processing platforms.

The specific contributions of our paper are:

- A simple task profiling method that provides sufficient information to determine processing requirements and traffic characteristics.
- A novel task duplication algorithm that determines the required number of task instances to avoid performance bottlenecks.
- A task allocation algorithm that assigns all task instances to processing resources while aiming for low communi-

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

cation overhead between processor cores.
- A prototype implementation of the system based on a modification to Click [12].

Our results show that the presented task allocation system is effective in managing processing resources and adapting to changes in workload. Our system can adapt in a single iteration, which is 5.3%–31.5% of the number of steps taken in iterative approaches. We also show that our runtime system can achieve 1.32× higher throughput for computationally demanding traffic than SMP Click [3], which is an adaptive packet processing system that uses general-purpose multi-core scheduling techniques.

The remainder of the paper is organized as follows. Section II discusses related work. Section III presents an overview of the runtime management system architecture. The task duplication algorithm, which is the central components of the runtime management system, is introduced in Sections IV. Section V presents the evaluation of our prototype system, and Section VI summarizes and concludes this paper. Additional discussion, including the task mapping algorithm we use in our work, can be found in supplementary material published together with this article.

II. Related Work

Flexibility in the data plane of next-generation networks has been explored in many different ways, ranging from new forwarding features [6] to network virtualization [1] and networks with dynamically configurable services [25]. All these approaches require a general software-programmable router platform, which is the focus for our work. General-purpose programmable packet processing systems have been developed in form of high-performance, general-purpose workstation processors (e.g., Sun's Niagara2 platform [10]), network processors (e.g., Intel's IXP platform [21]), and programmable logic devices (e.g., P4 [11] and NetFPGA [16]). In the context of next-generation Internet architectures, various router designs have employed specialized hardware to provide programmable packet processing functionality at high data rates (e.g., GENI backbone platforms [22] and high-performance PlanetLab nodes [23]). All these systems are characterized by featuring multiple parallel processing engines.

Most existing software development toolkits for parallel packet processors allocate processing tasks to processor cores automatically, but these allocations are static [8], [18]. Dynamic allocation mechanism have been proposed by Kokku et al. [13], where complete processors are turned on and off on demand, and in our prior work [26], where basic block allocated individually to processing cores. Neither of those systems uses a workload representation that is easily usable when developing packet processing systems (too coarse-grained, monolithic functions in [13] and too fine-grained instruction graph in [26]). A more suitable representation of data path processing is the Click modular router abstraction [12], which has also been expanded to be applicable for multiprocessor systems (SMP Click by Chen and Morris [3]) and network processors (NP-Click by Shah et al. [20]).

Recently, Kuang and Bhuyan developed a dynamic task allocation technique for network multi-processors [14]. This approach uses a parallelizing compiler to generate a task graph, which is then mapped to cores such that latency constraints are met and throughput is maximized. It can also be used to minimize power consumption [15]. The main difference to our work is that this approach does not use task duplication and thus needs to deal with tasks that can differ drastically in the amount of work require. Therefore, the task mapping process has a significantly higher computational complexity than our approach. While this overhead can be justified achieving near-optimal mappings, it may be less practical for environments with highly dynamics changes in traffic. In addition, neither [14] nor [15] consider task mapping at runtime. In our work, we develop a task mapping approach that dynamically adapts to changes in traffic.

The adaptation process in [13] uses the queue length of packets waiting to be processed as an indicator when more resources need to be allocated to a task. Similarly, the staged event-driven architecture (SEDA) described in [24] uses an adaptation algorithm based on response time to allocate processing resources. Both approaches used iterative adaptation of processing resources, whereas our approach directly computes the required allocation. Our approach is beneficial since changes in workload can be accommodated in a single adaptation step rather than multiple, iterative corrections. As our results show, the direct allocation approach leads to significantly fewer adaptation steps than an iterative approach, even if changes in workload only affect a small percentage of traffic.

Mallik and Memik have proposed to directly allocate processing tasks based on the statistics of their processing characteristics [17]. Their system allocates tasks statically to processors and uses an empirical approach to duplicating tasks to fill all processor cores. In our system, we continuously profile router tasks at runtime to dynamically reallocate tasks to processor cores. In addition, we use a task duplication approach that balances the workload more evenly to simplify the allocation process and achieve higher throughput. Our prior work [27], [28] discusses some initial ideas that have lead to the system presented here.

III. Runtime Management System Architecture

The main task of a runtime management system for packet processing hardware is to ensure that all processing resources are set up to process the packets that traverse the data path of the router. Changes in traffic characteristics require the system to adapt the configuration of the packet processing system at runtime. To implement such functionality, our runtime management system consists of several components that interact as illustrated in Figure 1. The main aspects are the offline programming and configuration step, the runtime management subsystem and the packet processing hardware. In this section, we describe each component in more detail to highlight their operations and interactions with other system components. Details on the specific algorithms used in each component are discussed in the following sections.

It is important to note that our system focuses on the runtime management of processing resources. While it is

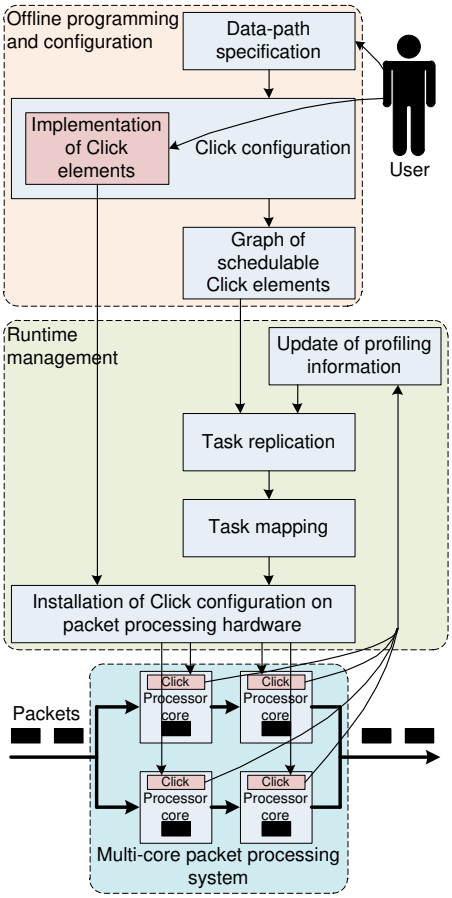


Fig. 1. Architecture of Runtime Management System for Multi-Core Packet Processing Systems.

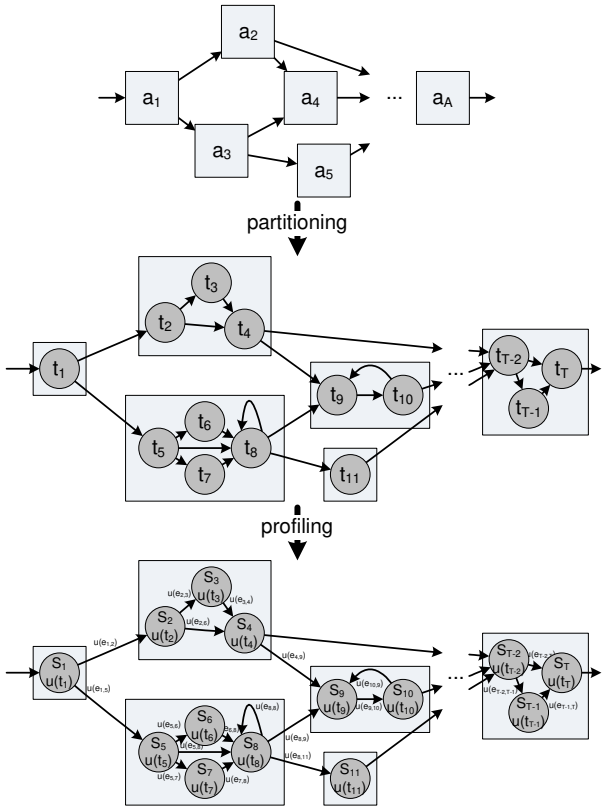


Fig. 2. Workload Partitioning and Profiling Steps.

also important to consider program data structures and their placement in memory, we do not consider this problem within the scope of this paper.

A. Programming and Configuration

The user (e.g., the administrator of the router system or the virtualization system managing the network) determines what functionality should be provided in the data path of the router. This "workload" needs to be represented in a suitable way such that it is (1) manageable for the user, (2) implementable on the underlying hardware, and (3) usable for the runtime system. Since no single representation can achieve all three goals, we use different workload models and automatically transition between them (as illustrated in Figure 1). For a more detailed discussion on this issue, see the supplementary material for this article.

The data path specification is typically represented as a graph (see top of Figure 2, where a_i represents a packet processing application). Network services are represented by nodes, and a directed edge indicates that there may exist some packets that require processing of the service from where the edge originates followed by the service 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.

When moving from the data path specification to the graph representation, each application needs to be partitioned into tasks (as illustrated in Figure 2). This step is necessary so that parallelism in multi-core packet processing systems can be exploited. One important question is how to partition data path operations. The programming language and development environment in which a network service is created often determines what level of partitioning is possible. For example, transition points between semantically separate processing steps in a network service (e.g., protocol header extraction, route lookup algorithm, checksum computation, etc.) could be used for partitioning. We chose to represent network services at the level of "tasks," where tasks are basic functional blocks in the network service similar to Click modules.

It is important to note that for our system practically *any* partitioning into tasks is suitable. In particular, the amount of processing associated with tasks does not need to be evenly distributed. With runtime profiling and task duplication, the runtime system can automatically adapt and compensate for differences. To transition from the task representation to the Click representation, we can use a one-to-one mapping between tasks and Click elements. Task dependencies are represented by communication links and queues in Click.

B. Runtime Profiling

The processing demands on the packet processing system are affected by two factors: first, by computational characteristics of all tasks in the system; second, by network traffic that exercises the processing system. In order to derive an optimal

allocation of tasks to processing resources at runtime, both factors 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 profiling information:

- Task Service Time s_i: For each task t_i, we determine the service time s_i (measured, for example, in number of processing cycles 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.
- Task Utilization $u(t_i)$: Based on usage counters, 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 graph representation of the workload with execution time distributions S_i and utilization $u(t_i)$ for each task. This is also illustrated in Figure 2. Since task utilization changes over time, we denote it as dependent on time parameter τ: $u^{\tau}(t_i)$. This time-dependence is further considered in Section III-D, 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).

Note that the task service time and task utilization can be obtained easily in a practical system. A packet processor can be augmented to count the number of times, n_i, a task processes packets over a window of time, δ, as well as the total amount of processing time spent, P_i. The processing time is simply sum of the execution time of each time a packet is processed by task s_i: $\sum_{j=1...n} s_i^j$. Using n_i and P_i, the service time can be estimated as $E[S_i] = P_i/\delta$ and the utilization as $u_i = n_i/\delta$. This process only requires the use of two registers on each processor core. With a sufficiently large δ, the overhead for moving profiling information from the core to the runtime management system can be kept to a minimum. Thus, obtaining the necessary profiling information for our runtime system can be implemented practically on packet processing systems.

C. Task Duplication and Mapping

Once profiling information is available, the runtime system can determine which tasks are particularly processing intensive. These tasks present potential performance bottlenecks if they cause processor cores to become overly loaded and trigger stalls in the software pipeline of the packet processing system. In conventional packet processing systems, the imbalance in processing time of tasks can be addressed by two approaches:

- Tasks can be modified to obtain a more balanced service time. This process typically requires the user/programmer to change the implementation of the tasks, which is difficult. Even if it is possible to automate the re-partitioning process, the drawback is that the semantic boundaries between tasks blur (i.e., protocol processing steps may be broken into multiple components).
- Tasks mapping can be performed with unbalanced tasks. In such an approach, tasks with high processing requirements are combined with tasks with low processing requirements on the same processor core (e.g., in separate hardware threads). The combination of high and low processing requirements can achieve a balanced processing load on the core. The drawback of this approach is twofold: (1) The problem of finding a good combination of tasks is equivalent to solving the bin packing problem, which is known to be intractable. It was noted in [9] that the imbalance of individual processing workloads is the major obstacle that keeps heuristic solutions from getting optimal mapping results. (2) Since the combination of tasks on a core is based on their processing requirements rather than on proximity of tasks in the packet flow through the system, high internal communication overhead may ensue.

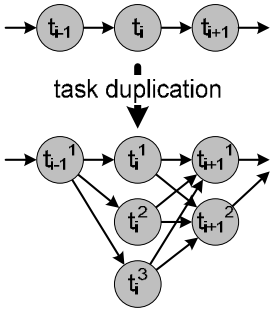


Fig. 3. Task Duplication Example with d_{i-1}=1, d_i=3, and d_{i+1}=2.

We present a novel approach to tackle these problems. We neither change the computational requirements of an individual task nor require the solving of a packing problem. Instead, we exploit that packet processing operations are (nearly) independent among packets of different flows and thus allows us to create additional task instances for processing-intensive tasks. Using this idea of *task duplication*, we create a task graph with d_i instances of each task t_i. As illustrated in Figure 3, each task instance is connected to all instances of preceding and succeeding tasks. When distributing packets evenly among all task instances, the utilization of each instance is reduced to u_i/d_i. Thus, we can use d_i to control how much processing is performed by each task. In Section IV, we present an algorithm for determining a d_i for all tasks such that processing requirements are nearly balanced across all instances. This balance of processing requirements among all instances then simplifies the task mapping problem as it removes the need to solve a packing problem. Since all tasks instance require nearly the same amount of processing, any allocation with an equal number of instances per processor core avoids performance bottlenecks.

With the ability to allocate tasks instances to processors nearly arbitrarily, a secondary metric can be employed to

optimize the allocation. In our system, we consider locality (i.e., 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 or cache. If the tasks reside on different processors, the processor interconnect needs to be used for the transfer. An algorithm to find a suitable mapping is described in the supplementary material.

D. Dynamic Adaptation

After a configuration of duplicated tasks and their mapping to processors has been determined and installed in the data path of the router, the runtime system needs to adapt dynamically. Dynamic adaptation 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, the task duplication and task mapping results need to adapt to changes in traffic.

The adaptation process used in our system is shown as Algorithm 1. For the system to adapt to changes in traffic, it is necessary to monitor traffic and its load on the system. For monitoring, the system profiles utilization parameter $u_\tau(t_i)$ for each task as an indicator of input traffic change. (It is also possible to monitor the service time distribution S_i in case processing demands change, but for our discussion, we assume S_i does not change over time.) If the difference between the current $u_\tau(t)$ and the previous $u_\tau(t-1)$ exceeds a threshold, ϵ, a new configuration is computed. First, the new number of task duplicates, d, is determined and then the mapping of tasks to threads, m. Both steps are explained in more detail below. Once the new mapping has been computed, the system switches from the previous task-to-processor allocation to the new one. The adaptation process is continued indefinitely at a fixed interval.

Algorithm 1 Runtime Adaptation Process.

```
1: function runtime_adaptation()
2: while true do
3:     if (difference(u_tau(t),u_tau(t-1))> ε) then
4:         d ← task_duplication()
5:         m ← map()
6:         commit_configuration(d,m)
7:     end if
8:     pause(interval)
9: end while
```

There are several implementation concerns with this approach. First, profiling should not cause too much overhead. It is possible to obtain $u_\tau(t_i)$ and S_i by simply maintaining two counters. One counter increments every time a task receives a new packet (i.e., reflecting the utilization), and one counter counts the total processing time spent by a task (i.e., reflecting the cumulative service times). Whenever the runtime adaptation system needs to update its value of $u_\tau(t_i)$ and S_i, it simply reads these two counter values from each task. A second concern is the impact of duplication and mapping computation on the forwarding performance of the system. We assume that the processing resources used for this computation are isolated from the data path of the router. For example, most modern network processors have a dedicated control processor that can be used for the runtime adaptation computation. Third, a new mapping of tasks may change the flow of traffic through the system. Thus, it is necessary to drain existing packets before reconfiguring the task allocation. This process may require that new packets be queued briefly until the system has been changed to the new configuration. Processing state may be migrated via shared memory (e.g., as discussed in [29]). Since we expect adaptation only to occur at time scales of seconds and the adaptation itself only takes milliseconds, we do not expect this to pose a performance bottleneck.

With this overview of the runtime system's operation, we turn to a more detailed discussion of its key elements (application partitioning and task duplication) in the following sections. Task mapping is discussed in supplementary material.

IV. Task Duplication

Task duplication is a key aspect of our runtime system. In order to fully utilize the available system resources and thus support the highest possible data rate, we need to consider the load that tasks place on the system's processing resources. A task's processing requires not only depend on how computationally demanding the 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:

$$w_i = u(t_i) \cdot E[S_i]. \tag{1}$$

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 using different workload representations that partition tasks differently. Task duplication is then used to decrease imbalance of individual tasks' work.

A. Duplication Process

We can adapt the amount of work of a task by changing its utilization through duplication. With a higher number of duplications, the utilization of each task instance decreases. 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:

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

B. 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 practical constraints. In all systems, there is a limit on the number of tasks instances that can be supported. Software-based systems have limits on the number of software threads per processor (to limit scheduler complexity) and hardware-based systems (e.g., network processors) have limits on the number of hardware threads per core. We denote the number of processors with N and the number of threads per processor with M. Thus, the duplication constraint is:

$$\sum_{i}^{T} d_i \leq N \cdot M. \quad (3)$$

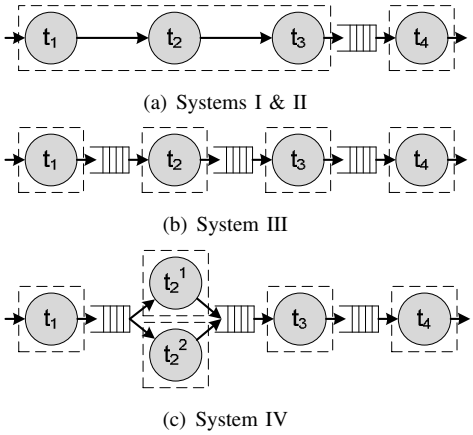
To make duplication choices that observe this resource constraint, we use a greedy approach shown as Algorithm 2. While processing resources are available, we identify the task that has the highest w'_i value. Adding a duplicated task instance to this task reduces the amount of work done by each instance 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 2 Task Duplication Algorithm.

```
1: function task_duplication()
2: for i = 1 to T do
3:     d_i ← 1
4:     w'_i = u(t_i)/d_i · E[S_i]
5: end for
6: while Σ_{i=1}^{T} d_i < N · M do
7:     j ← argmax_i w'_i
8:     d_j ← d_j + 1
9:     w'_j ← u(t_j)/d_j · E[S_j]
10: end while
11: return d
```

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., $T > N \cdot M$). This scenario is not very likely since even low-end network processor support high levels of multi-threading and Click applications do not require a large number of elements. However, if such a case occurs, then either tasks need to be merged 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 $T \leq N \cdot M$.

The duplication algorithm always seeks to assign tasks to all threads in the system. While this approach yields a good balancing of processing load, it may be the best choice in cases where the system is only lightly loaded. If the overall system is only lightly loaded, other optimization criteria can be considered (e.g., response time as proposed in [24] or jitter). Alternatively, some processor cores can be turned off to save power (e.g., as proposed in [13]).



(a) Systems I & II

(b) System III

(c) System IV

Fig. 4. Example Task Graphs for Different Systems. Schedulable units are separated by queues and denoted by dashed lines. (`Unqueue` elements are omitted for simplicity.)

V. Evaluation

To show the effectiveness of the proposed runtime system and its duplication and mapping algorithms, we present evaluation results from a prototype implementation.

A. Prototype System

Before discussing results, we first describe the prototype system, its workloads, and the testbed setup.

1) System Configurations: To evaluate the impact of runtime adaptation and task duplication, we compare several variations of our proposed router system with a traditional Click router. The systems used in our evaluation are:

- System I (Original Click on Unicore Processor [12]): This configuration uses unmodified Click source code.
- System II (Original Click with SMP Support [3]): This configuration uses unmodified Click source code that supports parallelism in form on multiprocessors. Both System I and II use the queue allocations determined by the original Click model. As illustrated in Figure 4(a), this leads to schedulable units that may contain multiple Click elements. These units all need to be allocated to the same processor.
- System III (Runtime Adaptation without Task Duplication): This configuration uses a version of Click that we have modified to separate all elements by queues. As illustrated in Figure 4(b), each element can be allocated separately to a different processor. This system does not use task duplication (to allow a quantitative evaluation of the overhead from profiling and additional queues). Runtime adaptation occurs in form of changes to task mapping when task utilizations change.

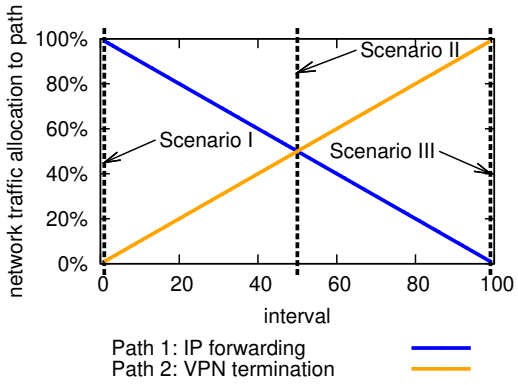


Fig. 5. Traffic and Workload Scenarios Used in Evaluation.

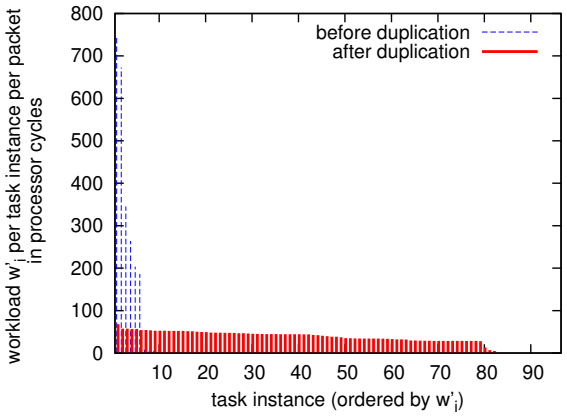


Fig. 6. Distribution of Work w_i' per Task Instance Before and After Duplication (Scenario I).

- System IV (Runtime Adaptation with Task Duplication): This configuration is based on System III, but does use task duplication (as illustrated in Figure 4(c)). Runtime adaptation occurs in form of changes to the number of duplicated tasks instances and changes to the task mapping. This configuration represents the entire functionality proposed in this paper.

In most cases, we compare Systems IV and II. System II (SMP Click [3]) represents the current state of the art of packet processing with general multiprocessor support. System IV represents the concept of task duplication and the associated task mapping that we present in this paper.

2) Traffic and Processing Workload: The complete Click router configuration that we use for this evaluation is shown in supplementary material. The configuration offers two choices on how packets are processed via different applications in the data path: (1) Path I: conventional IP forwarding and (2) Path II: VPN termination with IP forwarding. Path II represents a processing-intense data path since VPN termination requires cryptographic processing of the IPSec packet payload [19]. Thus, the overall processing demand is roughly proportional to the amount of traffic sent via Path II. To control the amount of traffic that traverses each path, we use the `StrideSwitch` element to direct packets in different proportions along each path. Our Click configuration consists of a total of 23 elements (i.e., tasks that can be duplicated and scheduled).

To explore network traffic with varying processing requirements, we use the workload illustrated in Figure 5. Traffic initially is mostly allocated to Path I and then transitions to Path II (i.e., from low processing requirements to high processing requirements). This transition happens in steps (i.e., intervals), and runtime adaptation is performed every time traffic changes. For experiments that compare our system to Click and SMP Click, we need to pick fixed traffic profiles. For these cases, we have identified three specific scenarios (also illustrated in Figure 5): Scenario I (99% IP forwarding, 1% VPN termination), Scenario II (50% IP forwarding, 50% VPN termination), and Scenario III (1% IP forwarding, 99% VPN termination).

3) Testbed Setup: The hardware platform for our testbed is a four-core Intel development board that is specifically designed for embedded systems. It is equipped with two 2.0 GHz Dual-Core LV Xeon processors and two dual port Intel 82571EB Gigabit Ethernet Controllers. The software is based on a modified Click release (version 1.6.0) running on a patched Linux kernel (version 2.6.16.13). Click is configured as kernel module and handles all packets transferred through the systems network interfaces. Four workstations connected to the system transmit and receive network traffic at varying data rates and packet sizes.

4) Representativeness of Evaluation: It is important to note that the results obtained from this experimentation configuration are representative for systems beyond this specific setup. While we use a general-purpose multicore processor system, the results are equally applicable to network processor systems. Click has been ported to network processors [20], and network processor cores that are ANSI C programmable have been developed [4].

In terms of processing workload, we only consider IPSec as a computationally intensive application (i.e., Path II). However, the results are representative of any type of packet processing. The runtime system does not distinguish between what operations are actually performed on a packet. As the Internet moves toward more data path services, Scenario III is becoming more representative.

B. Validation of Correct Operation

To show the correct operation of our duplication algorithm, we compare the work per task before duplication, w_i, with the work per task after duplication, w_i' in Figure 6 (for Scenario I and $M \cdot N$=96). The distribution of work per task instance before duplication is clearly unbalanced with tasks requiring between 5 cycles and 746 cycles per packet (155× difference). After duplication, these differences are much less pronounced. Task instances require between 5 cycles and 68 cycles (13.5× difference). While duplication reduces the difference between tasks significantly, the results are not a completely even distribution. The remaining imbalance is due to a few tasks with very small w_i values (that cannot be increased by duplication). When ignoring the three tasks with smallest w_i, the difference between the largest and smallest w_i' is only 2.43× after duplication.

TABLE I
SYSTEM UTILIZATION FOR DIFFERENT NUMBERS OF PROCESSORS (N) AND TOTAL NUMBER OF THREADS ($N \cdot M$)

Number of Processors (N)	Number of threads ($N \cdot M$)				
	32	64	128	256	512
2	88.37%	95.16%	97.19%	99.85%	99.44%
4	82.15%	92.45%	95.48%	93.45%	98.60%
8	50.55%	74.90%	86.34%	92.42%	96.63%
16	49.38%	72.36%	86.28%	92.43%	96.14%
32	47.17%	70.74%	86.28%	92.43%	96.14%

Note that for the scenario shown in the figure, 14 of the 23 tasks did not receive any packets (since they are associated with ARP processing) and thus their utilization (and work) is zero. Therefore, the 14 rightmost task instances have zero work associated with them. Still, each instance is present in the system to ensure that all packets can be correctly processed in case traffic changes.

One key question in the context of duplication is what value M, the number of threads per processor, should be. The higher M, the more balanced the overall workload. However, very high values of M create overhead in the scheduling system of Click and are thus undesirable. Therefore, we explore the question of how well we can utilize a system with limited M for different numbers of processor cores, N. The results are shown in Table I where the system utilization for different configurations is shown. A higher utilization indicates that the systems performs better under that configuration. For small N, the balance is near optimal. For larger numbers of processors, a larger number of threads are necessary to achieve balance. Even for just four cores, it is desirable to allow a total of over 64 tasks in order to get a utilization that is above 90%. For high numbers of threads, the system achieves very high utilization of over 96%.

C. Comparison to Iterative Adaptation

As discussed in Section II, the adaptation approaches described in [13] and [24] are based on monitoring queue lengths and response time. In response to increasing queue lengths or increasing response time, these approaches allocate additional resources and thus can adapt to changing workloads. A key difference to our duplication approach is that these approaches changed allocations *iteratively*. If there is a significant change in the amount of traffic going to a particular tasks, these approaches cannot immediately determine how many additional resources are necessary. Instead, multiple adaptation steps are necessary until enough resources have been allocated and queue lengths no longer grow or response time requirements are met. In contrast, duplication *explicitly* computes the number of tasks necessary for handling a certain workload based on $u(t_i)$.

To demonstrate that there are significant benefits to the duplication approach, we present a comparison of the number of adaptation steps that are necessary as workload changes. We consider a workload change from Scenario I to Scenario III with different step sizes (i.e., 1% steps, 2% steps, etc.). Table II shows the average and maximum number of adaptations that are necessary to accommodate these changes for both types of approaches. The results show for larger workload changes, the number of adaptation steps in an iterative process can be very large. This implies a larger overhead for adaptation since the system needs to reconfigure multiple times. In contrast, duplication can determine how to adapt in a single step. (The average for duplication is below one since some changes do not require changes in task allocations.) These results show that there is great benefit in using duplication over iterative adaptation since it makes the system more responsive.

TABLE II
NUMBER OF ADAPTATION STEPS OF ITERATIVE APPROACHES VS. DUPLICATION

Workload change	Iterative		Duplication		Fraction (for avg.)
	avg.	max.	avg.	max.	
1%	2.16	37	0.68	1	31.5%
2%	3.65	63	0.71	1	19.5%
3%	5.19	79	0.88	1	17.0%
5%	8.32	97	0.79	1	9.5%
10%	17.22	121	0.89	1	5.2%

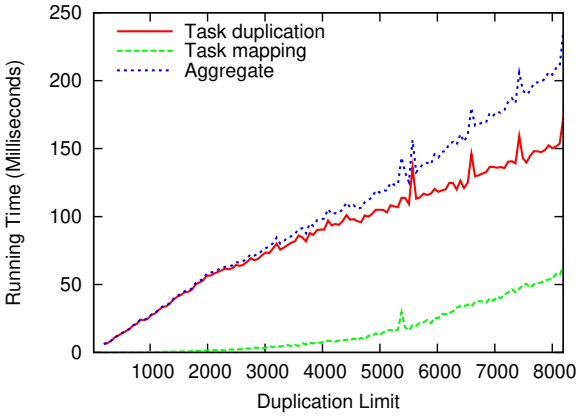


Fig. 7. Processing Time Requirements for Task Duplication and Task Mapping Algorithms for Varying Duplication Limit (i.e., number of overall threads $M \times N$).

D. Processing Time for Adaptation

The analysis of the computational complexity of the algorithms used in our work is provided in supplementary material. The measured processing times of the prototype task duplication and task mapping algorithms are shown in Figure 7. The number of tasks is $T = 23$. As expected, for task duplication we can observe a general trend that is approximately linear with the number of task instances, $M \cdot N$. For low duplication limits, the executing time is slightly below the extrapolated slope since the algorithm runs faster while it still fits into the processor's cache. For task mapping, the initially linear trend gains an upward slope as large number of duplications increase the number of edges, E. Overall, the results show a roughly linear trend for the aggregate processing time. Even for large numbers of task instances, the processing time is very fast (e.g., around 200 milliseconds for $M \cdot N$=8000.).

For practical systems, where the task duplication limit is fixed, the aggregate processing time puts a lower bound on the adaptation interval. The execution time (and thus the interval

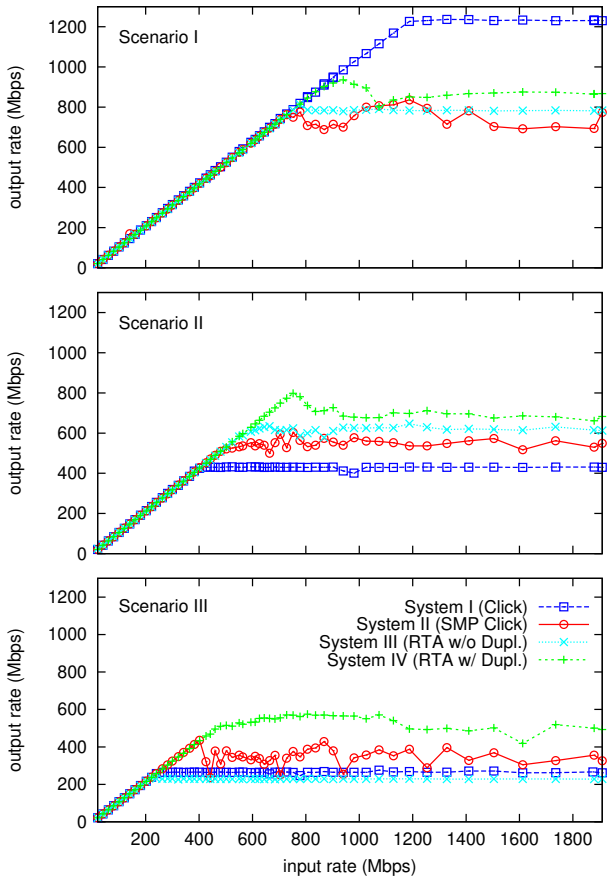


Fig. 8. Throughput Performance Comparison (RTA = Runtime Adaptation).

bound) may be higher if the same processor is used for computing task duplication and mapping and for forwarding packets. In a high-performance system, however, runtime management is typically performed on a dedicated control processor.

E. Performance Comparison

To compare the performance of all four systems discussed above, we show their throughput performance for all three scenarios in Figure 8. Since we focus on the processing aspect of the router (rather than its ability to get packets in and out of the system), we send large (1452-byte) UDP packets. (Results for minimum size packets are discussed below.) In Scenario I, there is very little processing, but in Scenario III, all packet payloads need to be processed by cryptographic algorithms. We make the following observations:

- Scenario I: This scenario, which has very low computational demands, is dominated by System I, the traditional Click unicore system, which achieves over 1.2Gbps in throughput. The main limitation for other systems, which utilized multiple processor cores, is the overhead for packet I/O and coordination among processors. Systems II, III, and IV can forward between 700Mbps and 900Mpbs. Our proposed System IV performs slightly better than Systems II and III, but its full capabilities cannot be shown in a scenario that has limited processing demands.

TABLE III
PEAK THROUGHPUT RATES FOR ALL ROUTER SYSTEMS.

Scenario	Pkt. size in bytes	Peak throughput in Mbps I	II	III	IV	Improvement of System IV over II
I	64	57	44	33	39	-12.0% (=0.88×)
	1452	1237	837	813	936	11.8% (=1.11×)
II	64	66	61	48	49	-19.7% (=0.80×)
	1452	432	603	647	798	32.2% (=1.32×)
III	64	72	56	45	48	-13.9% (=0.86×)
	1452	276	435	237	575	32.0% (=1.32×)

- Scenario II: In this scenario, where more processing is required, traditional Click (System I) starts to perform poorly since it uses only a single processor core (around 400Mbps). Our proposed System IV achieve around 800Mbps at its peak. SMP Click (System II) levels out at 600Mbps.
- Scenario III: This scenario requires the most amount of processing per packet. In this case, our proposed System IV achieves nearly 600Mbps, whereas SMP Click (System II) peaks out at just above 400Mbps.

The peak rates are summarized in Table III. This table also shows throughput results for small packets. For small packets, the proposed System IV does not perform as well as SMP Click (between 12% and 20% lower throughput). This is due to the overhead of task duplication, which leads to more packet movement between processors. However, small packets require much less processing in Path II since their payload is minimal. Therefore, such traffic (similar to Scenario I) is not representative of processing-intensive workloads for which we target the design of System IV. Instead, Scenarios II and III with larger packets should be considered the main evaluation target for our platform. As the Internet moves toward more data path services (at least on the network edge), such scenarios will become more common. Also, System IV performs better than System II for packet size larger than around 134–146 bytes, which indicates that System IV outperforms System II for typical packet size distributions found in the Internet.

For these processing-intensive scenarios, our runtime adaptation system with task duplication (System IV) can sustain a 1.32× higher data rate than SMP Click (System II). Also, our system performs consistently well for all three scenarios and does not show the severe degradations observed in Click (System I). This indicates that our runtime system uses available processing resources more effectively. Thus, we can conclude that our approach of task duplication and mapping presents a significant improvement to runtime management of parallel packet processing systems.

VI. SUMMARY AND CONCLUSION

The need for flexibility and performance in the data path of routers motivates the need for packet processing systems that utilize multiple parallel processor cores. To provide the ability to adapt to changing traffic characteristics, it is essential to have a runtime management system in place that can adjust the allocation of processing tasks to processing resources. We present a runtime system that uses runtime profiling and a novel task duplication algorithm to achieve a balanced

workload across all processor cores in a single iteration. We present a prototype implementation of this runtime system that is based on the Click modular router. Our extensive evaluation of this system shows the correct operation of the algorithms. Our results also show that our runtime system requires only 5.3%–31.5% of the adaptation steps of iterative approaches and outperforms SMP Click by providing 1.32× higher throughput for processing intensive packets. We believe that the concepts introduced in this runtime system present an important step toward implementing and deploying highly parallel packet processing systems in the current Internet and next-generation network architectures.

ACKNOWLEDGEMENTS

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

REFERENCES

[1] 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.

[2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from Berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec. 2006.

[3] B. Chen and R. Morris, "Flexible control of parallelism in a multiprocessor PC router," in *Proc. of the General Track: 2002 USENIX Annual Technical Conference*, Monterey, CA, Jun. 2001, pp. 333–346.

[4] *The Cisco QuantumFlow Processor: Cisco's Next Generation Network Processor*, Cisco Systems, Inc., San Jose, CA, Feb. 2008.

[5] D. D. Clark, "The design philosophy of the DARPA Internet protocols," in *Proc. of ACM SIGCOMM 88*, Stanford, CA, Aug. 1988, pp. 106–114.

[6] 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.

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

[8] 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.

[9] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, Mar. 1969.

[10] G. Grohoski, "Niagara2: A highly threaded server-on-a-chip," in *Proc. of Symposium on High Performance Chips (HOT CHIPS 18*, Palo Alto, CA, Aug. 2006.

[11] I. Hadzic, W. S. Marcus, and J. M. Smith, "On-the-fly programmable hardware for networks," in *Proc. of IEEE Globecom 98*, Syndey, Australia, Nov. 1998.

[12] 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.

[13] 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.

[14] J. Kuang and L. Bhuyan, "LATA: a latency and throughput-aware packet processing system," in *Proceedings of the 47th Design Automation Conference (DAC)*, Anaheim, CA, Jun. 2010, pp. 36–41.

[15] ——, "Optimizing throughput and latency under given power budget for network packet processing," in *Proceedings of the 29th conference on Information communications (INFOCOM)*, Mar. 2010, pp. 2901–2909.

[16] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo, "NetFPGA–an open platform for gigabit-rate network switching and routing," in *MSE '07: Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education*, San Diego, CA, Jun. 2007, pp. 160–161.

[17] 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.

[18] W. Plishker, K. Ravindran, N. Shah, and K. Keutzer, "Automated task allocation for network processors," in *Proc. of Network System Design Conference*, Oct. 2004, pp. 235–245.

[19] R. Ramaswamy, N. Weng, and T. Wolf, "Analysis of network processing workloads," *Journal of Systems Architecture*, vol. 55, no. 10, pp. 421–433, Oct. 2009.

[20] N. Shah, W. Plishker, K. Ravindran, and K. Keutzer, "NP-Click: A productive software development approach for network processors," *IEEE Micro*, vol. 24, no. 5, pp. 45–54, Sep. 2004.

[21] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb, "Building a robust software-based router using network processors," in *Proc. of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, Banff, AB, Oct. 2001, pp. 216–229.

[22] J. S. Turner, "A proposed architecture for the GENI backbone platform," in *Proc. of ACM/IEEE Symposium on Architectures for Networking and Communication Systems (ANCS)*, San Jose, CA, Dec. 2006, pp. 1–10.

[23] 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.

[24] M. Welsh and D. Culler, "Adaptive overload control for busy internet servers," in *Proc. of the 4th conference on USENIX Symposium on Internet Technologies and Systems (USITS)*, Seattle, WA, Mar. 2003.

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

[26] 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, Jul. 2007.

[27] 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.

[28] ——, "On runtime management in multi-core packet processing systems," in *Proc. of ACM/IEEE Symposium on Architectures for Networking and Communication Systems (ANCS)*, San Jose, CA, Nov. 2008, pp. 69–78.

[29] ——, "Support for dynamic adaptation in next generation packet processing systems," in *Proc. of IEEE International Conference on Communications (ICC)*, Dresden, Germany, Jun. 2009.

Qiang Wu received an M.E. degree in electronic and communication engineering from Tsinghua University, China, in 2006 and a Ph.D. in Electrical and Computer Engineering from the University of Massachusetts Amherst in 2010. He currently is a Member of Technical Staff at Juniper Networks, Inc. His research interests are network system design, performance analytical model, and parallel processing.

Tilman Wolf (M'02,SM'07) received a Diplom in informatics from the University of Stuttgart, Germany, in 1998. He also received a M.S. in computer science in 1998, a M.S. in computer engineering in 2000, and a D.Sc. in computer science in 2002, all from Washington University in St. Louis. Currently, he is an associate professor in the Department of Electrical and Computer Engineering at the University of Massachusetts Amherst.

He is engaged in research and teaching in the areas of computer networks, computer architecture, and embedded systems. His research interests include network processors, their application in next-generation Internet architectures, and embedded system security.

Dr. Wolf is a senior member of the IEEE and member of the ACM. He has been active as program committee member and organizing committee member of several professional conferences, including IEEE INFOCOM and ACM SIGCOMM. He is currently serving as treasurer for the ACM SIGCOMM society and on the editorial boards of the IEEE/ACM Transactions on Networking and IEEE Micro.

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

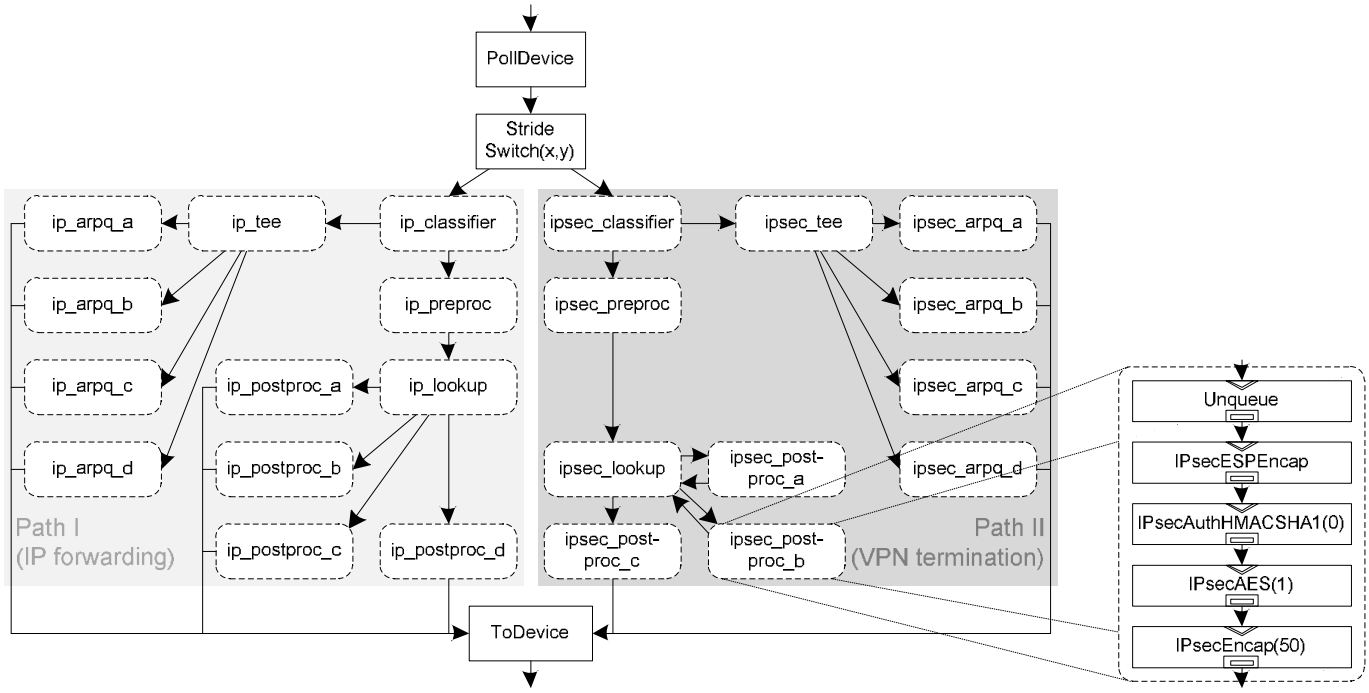


Fig. 1. Tasks in Click Router Configuration Used in Evaluation. Tasks contain multiple Click elements (as shown for `ipsec_postproc_b`).

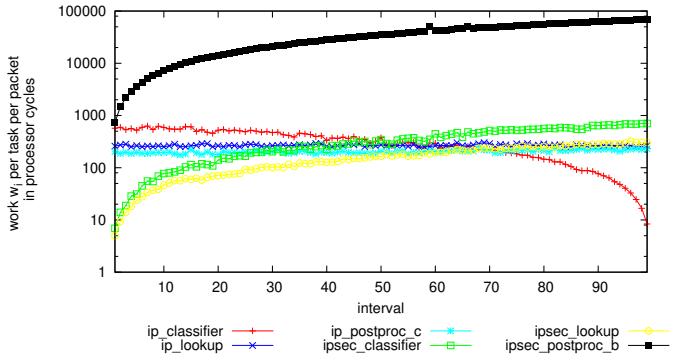


Fig. 2. Work w_i of Six Tasks over Different Intervals.

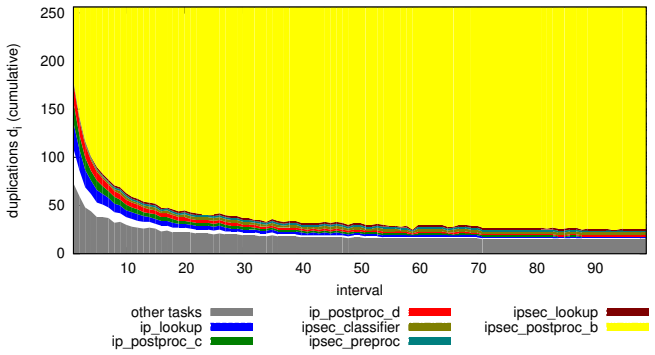


Fig. 3. Cumulative Duplications for Tasks over Different Intervals.

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.