On Runtime Management in Multi-Core Packet Processing Systems

Qiang Wu and Tilman Wolf
Department of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA 01003
{qwu,wolf}@ecs.umass.edu

ABSTRACT

Computer networks require increasingly complex packet processing in the data path to adapt to new functionality requirements. To meet performance demands, packet processing systems on routers employ multiple processor cores. We investigate the design of an efficient run-time management system that handles the allocation of processing tasks to processor cores. Using run-time profiling information about processing requirements and traffic characteristics, the system is able to adapt to dynamic changes in the workload and balance the utilization of all processing resources to maximize throughput. We present a prototype implementation of our system that is based on the Click modular router. Our results show that our prototype system can adapt to changing workloads and process computationally demanding packets at 1.32 times higher data rates than SMP Click.

General Terms

Design, Performance, Algorithms

Categories and Subject Descriptors

C.2.6 [**Computer-Communication Networks**]: Internetworking—*Routers*; C.1.4 [**Processor Architectures**]: Parallel Architectures

1. INTRODUCTION

The complexity of operations performed in the data path of today's Internet has expanded significantly beyond the simple store-and-forward concepts proposed in the original architecture. Packet processing steps in IP routers are numerous and include packet classification, content inspection, traffic shaping, and accounting. It can be expected that the trend towards more functionality and complexity in the data path continues in order to accommodate demands for more security features, operational controls, and new data path services [6]. A similar need for complex data path functionality can be observed in the proposed architectures for the next-generation Internet [7]: 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. A key challenge in the operation of parallel packet processing system is how to allocate a suitable workload to each processing engine. The difficulty of this process lies in several inherent properties of packet processing workloads. First, processing demands for different packet processing steps vary with changes in network traffic (e.g., IPSec processing varies with amount of VPN traffic). This variability requires packet processing systems to be flexible in terms of how many processing resources are allocated to a particular type of packet processing step. Second, embedded processing engines are tightly coupled to each other. If one engine is overloaded, it may present a bottleneck in the stream of packets traversing the packet processing system and thus cause other processing engines to perform poorly. Therefore, it is important that multi-core packet processing systems are carefully managed during runtime in order to translate their raw processing power into high packet throughput.

We address this question of how to manage multi-core packet processing systems in this paper. Starting from a Click representation of the router's data path operations, we use runtime profiling information to determine the processing requirement of each router module. Using a novel approach of *task duplication* according to processing demands, we can balance the amount of work each module needs to perform. This balance allows us to use a straightforward *task mapping* algorithm to assign processing tasks to processing resources while achieving global workload balance. As profiling information changes, the allocations can be updated during runtime. The specific contributions of our work are:

- Extension of the Click modular router to support runtime profiling and task duplication on multi-core platforms.
- Development of a novel task duplication algorithm and a task mapping algorithm that balances the workload across the system.
- Evaluation of a prototype implementation of our system and a comparison to SMP Click.

ANCS'08, November 6–7, 2008, San Jose, CA, USA.

Our results show that our runtime system can achieve 1.32× higher throughput for computationally demanding traffic than SMP Click, which is the current state-of-the-art adaptive packet processing system on general-purpose multi-core processors. It is important to note that our results are generally applicable to any type of multi-core packet processing system ranging from multi-core general-purpose processors to network processors to programmable logic devices.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 presents an overview of the system architecture and how packet processing workloads are represented. The task duplication and task mapping algorithms, which are the central components of the run-time management system, are presented in Section 4. Section 5 presents the evaluation of our prototype system, and Section 6 summarizes and concludes this paper.

2. RELATED WORK

Flexibility in the data path of networks has been explored in many different ways, ranging from extensible operating system kernels (e.g., x-Kernel by Hutchinson and Peterson [12]) to routers with selectable features (e.g., router plugins by Decasper et al. [5]) to active networks with programmable and customizable processing capabilities (e.g., capsules by Tennenhouse and Wetherall [20]) to routers fully customizable protocol stacks (e.g., router virtualization by Anderson et al. [1]). Our work assumes a software-programmable router platform, but could be extended to other systems with programmable data paths.

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 [19]), and programmable logic devices (e.g., P4 [11] and NetFPGA [15]). In the context of next-generation Internet architecture proposals [7], various router designs have employed specialized hardware to provide programmable packet processing functionality at high data rates (e.g., GENI backbone platforms [21] and high-performance PlanetLab nodes [22]). 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 [9, 17]. Dynamic allocation mechanism have been proposed by Kokku et al. [14], where complete processors are turned on and off on demand, and by Wolf et al. [23], 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 [14] and too fine-grained instruction graph in [23]). A more suitable representation of data path processing is the Click modular router abstraction [13], 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. [18]). Based on the Click, Mallik and Memik have recently proposed to allocate processing tasks based on the statistics of their processing characteristics [16]. Their system allocates tasks statically to processors and uses an empirical approach to duplicating tasks to fill all processor cores. In our system, we propose an approach to continually profile the Click router during run-time in order to dynamically reallocate

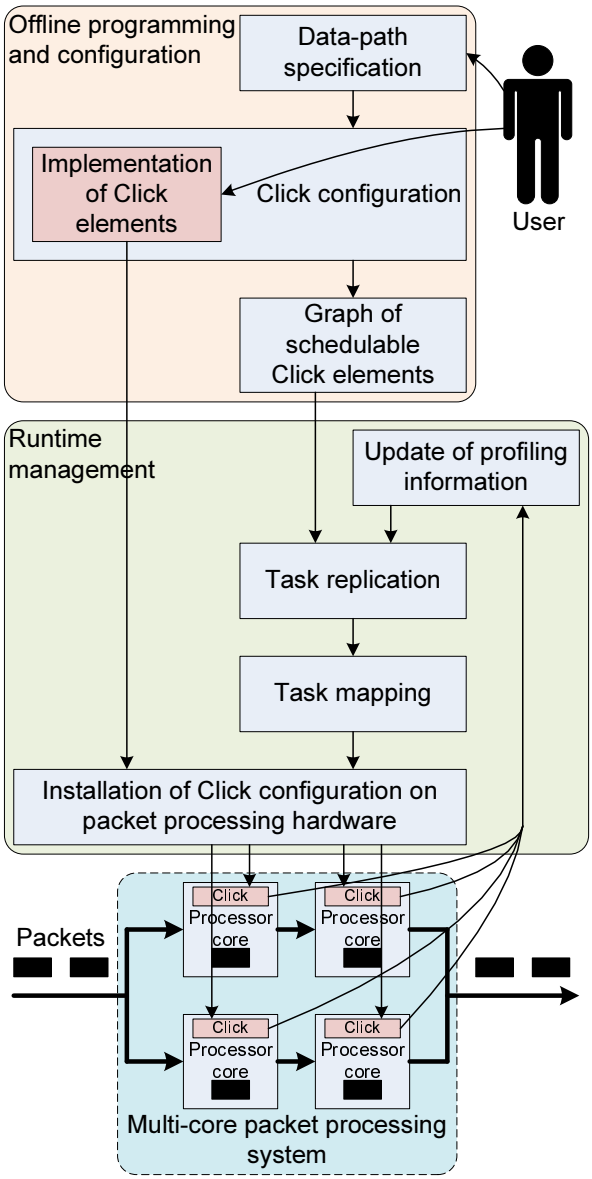


Figure 1: Architecture of Runtime Management System for Multi-Core Packet Processing Systems.

modules to processors. Also, we use a duplication approach that balances the workload more evenly as we allow multiple threads per processor.

3. 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 a number of components that interact as illustrated in Figure 1. The main components are the offline programming and configuration step, the runtime management subsystem and the packet processing hardware. In this section, we describe each com-

ponent in more detail to highlight their operation and interactions with other system components.

Is is important to note that our system focuses on the runtime management of processing resource. While it is also important to consider program data structures and their placement in memory, we do not consider this problem within the scope of our paper.

3.1 Programming and Configuration

The user (i.e., the administrator of the router system) determines what functionality should be provided in the data path of the system. 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 run-time 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).

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 [2, 8]. The data path specification determines which sequences of applications can be traversed by packets.

- 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.

- 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." 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.

When moving from the data path specification to the graph representation, each application needs to be partitioned into tasks. Application partitioning has been studied extensively in related work. For our system, it is important to note that practically *any* partitioning into tasks is suitable (e.g., simply selecting Click elements to represent tasks). In particular, the amount of processing associated with tasks does not need to be evenly distributed. As we show below, the run-time system can automatically adapt and compensate. 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.

Given this ability to move from a user-level representation of the router data path operations to a graph representation of the workload, we can now turn towards the question of how to profile, duplicate, and allocate tasks.

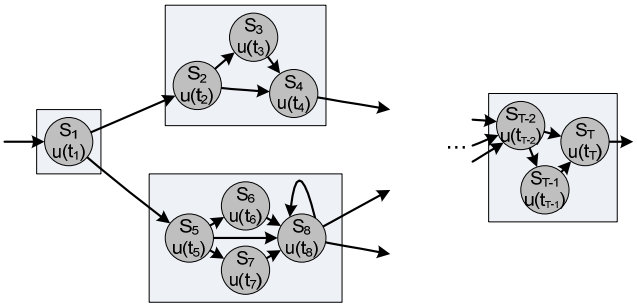


Figure 2: Graph Representation with Profiling Information.

3.2 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 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 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 3.4, 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).

3.3 Task Duplication and Task Mapping

Once profiling information is available, the runtime system can determine which tasks are particularly processing intensive. These tasks present an imbalance in the system and make it difficult to derive a task mapping that leads to high system utilization.

Since we cannot change the service time for a task (unless we ask the user/programmer to change the implementation,

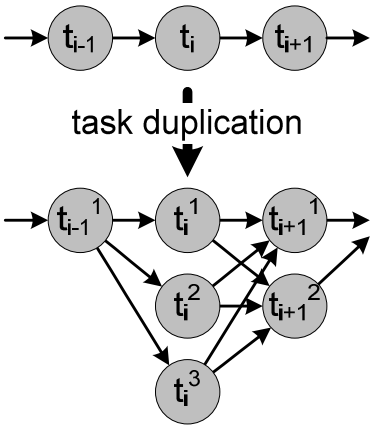


Figure 3: Task Duplication Example with d_{i-1}=1, d_i=3, and d_{i+1}=2.

which would be difficult), we use the idea of *task duplication* to adapt the processing demand of a task by changing its utilization. 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. This process is illustrated in Figure 3. We use parameter d_i to indicate the number of duplicated instances that exist for task t_i. Section 4.2 presents the algorithm that determines the optimal value of d_i.

With an extended graph representation that includes duplicated instances of schedulable Click elements, the runtime system needs to map each task to a hardware packet processing element. 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 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 more detail in Section 4.3.

3.4 Dynamic Adaptation

After a configuration has been installed in the data path, the runtime system continues to operate while adapting 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, we need adapt the task duplication and task mapping results to changes in traffic.

Since the profiling component monitors the dynamic trends in the processing workload, we can implement such adaption easily. Changes in traffic are reflected in the utilization parameters $u^\tau(t_i)$. Thus, duplication and mapping can be updated based on this information. 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 in this context (e.g., mapping based on length of inter-processor queues [14], mapping based on fixed intervals optimized for workload [23]).

4. DUPLICATION AND MAPPING ALGORITHMS

With an understanding of the overall system operation as described in the previous section, we now turn to the question of how the two key algorithms, task duplication and task mapping, work. To formalize the description of the algorithms, we start with a problem statement.

4.1 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 of resource limitations: $\forall j, 1 \leq j \leq N : |\{t_i | m(t_i) = j\}| \leq M$.

Processing resources are typically threads on a processor core. Most high-performance packet processing systems support hardware multi-threading, which presents an inherent limit on resources (i.e., M tasks per processor). In Click, tasks are scheduled in software and thus not inherently limited. However, practical concerns (e.g., scheduler complexity) also limit the maximum number of software tasks.

Tasks may be mapped to multiple processing resources. If tasks are computationally demanding, a single processing resource may not be sufficient and thus multiple processing resources could be used (i.e., task duplication). Since there are no inter-packet dependencies for most packet processing applications, such parallelization is easily possible.

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 (i.e., a mapping such that the difference between the maximum and minimum processor utilization across the system is minimized). The reasoning behind this metric is that such a mapping can provide the highest overall throughput without overloading any particular processor core. 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.

4.2 Task Duplication

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:

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

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 using different workload representations that partition tasks differently. Instead, it is necessary to dynamically balance the work for each task.

4.2.1 Duplication Process

We can adapt the amount of work of a task by changing its utilization through duplication. The higher the number of duplications, d_i, the lower the utilization. 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':

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

4.2.2 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 ∑_{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., $T > N \cdot M$). 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 $T \leq N \cdot M$.

4.3 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:

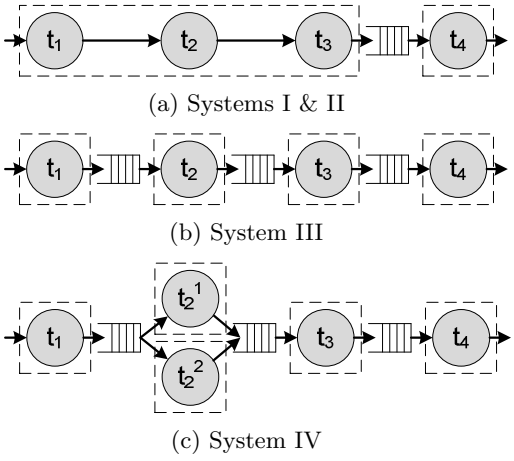
- Task Locality: Tasks t_i and t_j that are connected through an edge e_{ij} (or through a short path of edges), in practice often may share data structures. Thus, placing these tasks on the same packet processing engine may improve the efficiency of the system (e.g., caching is more effective, locks 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 2 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 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(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



(a) Systems I & II

(b) System III

(c) System IV

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

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

5. EVALUATION

To evaluate the effectiveness of the proposed runtime system, we have implemented a prototype.

5.1 Prototype System

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

5.1.1 System Configurations

In order to evaluate the impact of runtime adaptation and task duplication, we consider several variations of our proposed router system and a traditional Click router. The systems used in our evaluation are:

- System I (Original Click on Unicore Processor [13]): 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.
- 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.

Note that Systems III and IV require several small changes to Click. 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. In addition, we implement a blocking mechanism for `MSQueue` to prevent packets from being dropped when a queue is full (which would lead to a performance collapse under load). 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 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.

5.1.2 Traffic and Processing Workload

The Click router configuration that we use during this evaluation 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 IPSec processing. 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). For experiments that compare our system to Click and SMP Click, we need 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).

5.1.3 Testbed Setup

The hardware platform for our testbed is a four-core Intel

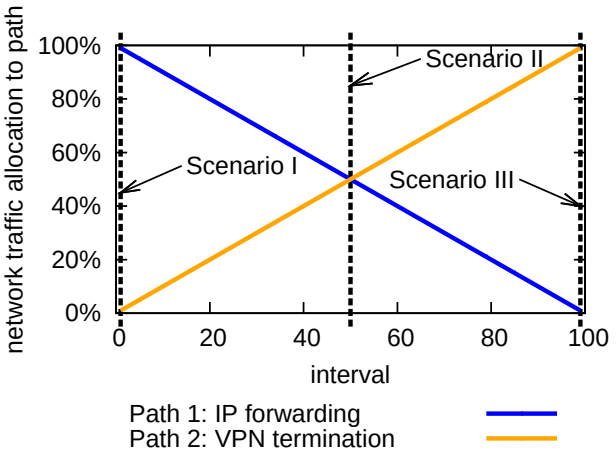


Figure 5: Traffic and Workload Scenarios Used in Evaluation.

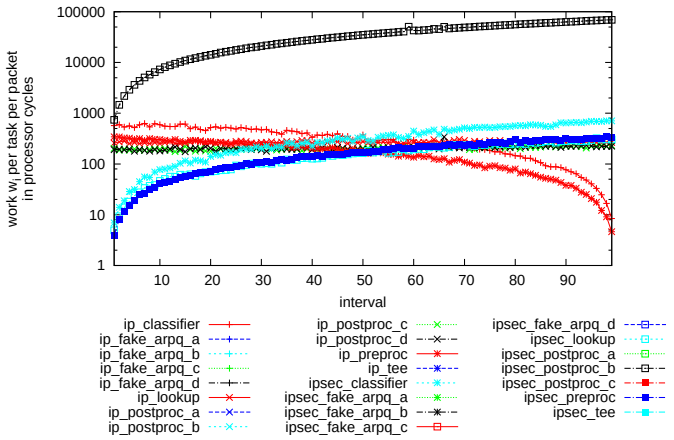


Figure 6: Work w_i of All 23 Tasks over Different Intervals.

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.

5.1.4 *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 [18], 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 towards more data path services, Scenario III is becoming more representative.

5.2 Validation of Correct Operation

Before comparing different system, we first show results that indicate that our proposed system (System IV) operates as described in Sections 3 and 4.

5.2.1 *Profiling*

To illustrate the functionality of the profiling component in our runtime system, we show in Figure 6 the profiling information for each of the 23 schedulable elements over the range of all processing workloads (Figure 5). The y-axis of this figure shows the total work per task per packet (as defined in Equation 1) 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.

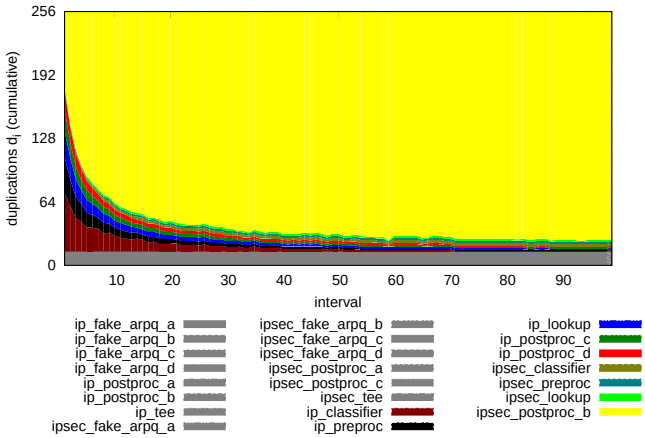


Figure 7: Cumulative Duplications for All 23 Tasks over Different Intervals.

5.2.2 *Task Duplication and Mapping*

The profiling information from Figure 6 is then used to duplicate tasks. Figure 7 shows the number of times each of the tasks is duplicated. 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`).

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 8 (for Scenario I and $M \cdot N$=96). For the interval 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.

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 du-

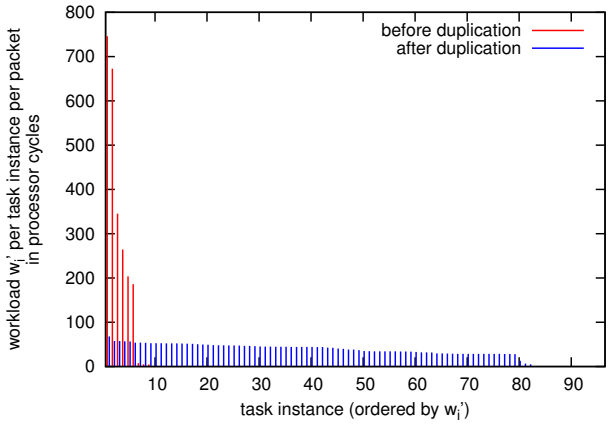


Figure 8: Distribution of Work w_i' per Task Instance Before and After Duplication (Scenario I).

Table 1: Processing Time Requirements for Task Duplication and Task Mapping Algorithms.

Number of tasks	Execution time
64	139ms
128	198ms
256	228ms

plication, 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.

5.2.3 *Runtime Adaptation Overhead*

The execution times of the task duplication and task mapping algorithm are shown in Table 1. Depending on the number of tasks that need to be handled, the processing time can take up to 0.2 seconds. This puts a lower bound on the adaptation interval. The execution time (and thus the interval 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 would be performed on a dedicated control processor.

5.3 Performance Comparison

To compare the performance of all four systems discussed above, we show their throughput performance for all three scenarios in Figure 9. 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 IV and II, but its full capabilities cannot be shown in a scenario that has limited processing demands.

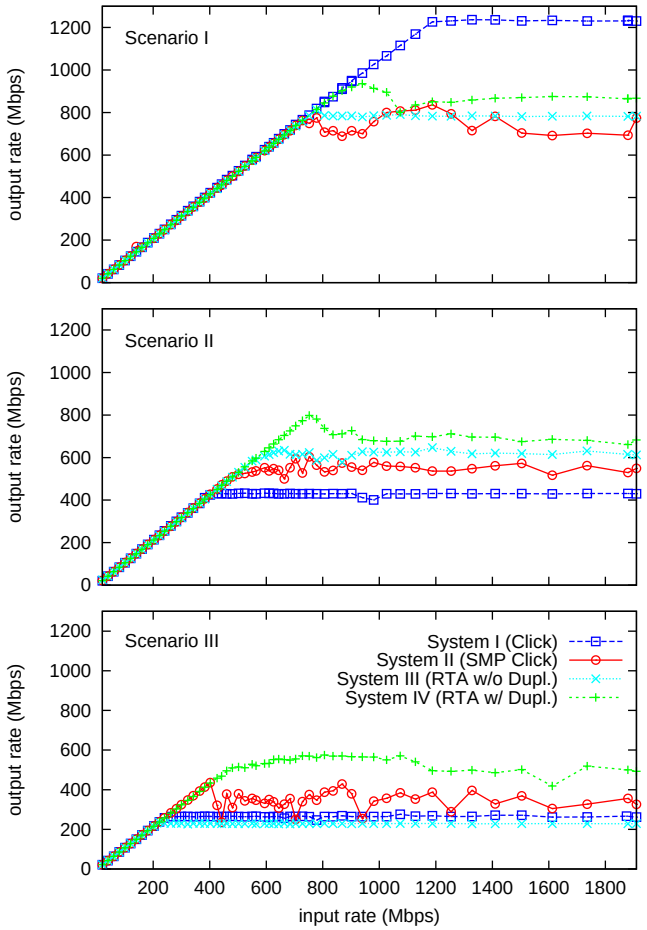


Figure 9: Throughput Performance Comparison. (RTA = Runtime Adaptation.)

- 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 2. This table also shows throughput results for small packets. For small packets, the proposed System IV does not perform as well as

Table 2: Peak Throughput Rates for All Router Systems.

Scenario	Packet size in bytes	Peak throughput (in Mbps) I	II	III	IV	Improvement of System IV over II
I	64	57.23	44.21	32.85	38.92	-12.0% (=0.88×)
	1452	1236.74	836.60	813.11	935.51	11.8% (=1.11×)
II	64	65.96	61.16	47.78	49.09	-19.7% (=0.80×)
	1452	432.38	603.26	646.78	797.78	32.2% (=1.32×)
III	64	72.33	56.10	45.28	48.33	-13.9% (=0.86×)
	1452	275.64	435.14	236.96	574.51	32.0% (=1.32×)

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 (similarly 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 towards more data path services (at least on the network edge), such scenarios will become more common.

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.

6. 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. 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 that we present. Our results also show that our runtime system outperforms SMP Click and provides 1.32× higher throughput for processing intensive packets. We believe that the concepts introduced in this runtime system present an important step towards implementing and deploying highly parallel packet processing systems in the current Internet and next-generation network architectures.

7. REFERENCES

[1] Anderson, T., Peterson, L., Shenker, S., and Turner, J. Overcoming the Internet impasse through virtualization. *Computer 38*, 4 (Apr. 2005), 34–41.

[2] Calvert, K. L., Griffioen, J., and Wen, S. 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.

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

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

[5] Decasper, D., Dittia, Z., Parulkar, G., and Plattner, B. Router Plugins - a modular and extensible software framework for modern high performance integrated services routers. In *Proc. of ACM SIGCOMM 98* (Vancouver, BC, Sept. 1998), pp. 229–240.

[6] Eatherton, W. 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] Feldmann, A. Internet clean-slate design: what and why? *SIGCOMM Computer Communication Review 37*, 3 (July 2007), 59–64.

[8] Ganapathy, S., and Wolf, T. Design of a network service architecture. In *Proc. of Sixteenth IEEE International Conference on Computer Communications and Networks (ICCCN)* (Honolulu, HI, Aug. 2007), pp. 754–759.

[9] Goglin, S. D., Hooper, D., Kumar, A., and Yavatkar, R. Advanced software framework, tools, and languages for the IXP family. *Intel Technology Journal 7*, 4 (Nov. 2003), 64–76.

[10] Grohoski, G. 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] Hadzic, I., Marcus, W. S., and Smith, J. M. On-the-fly programmable hardware for networks. In *Proc. of IEEE Globecom 98* (Syndey, Australia, Nov. 1998).

[12] Hutchinson, N. C., and Peterson, L. L. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering 17*, 1 (Jan. 1991), 64–76.

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

[14] Kokku, R., Riché, T., Kunze, A., Mudigonda, J., Jason, J., and Vin, H. 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).

[15] Lockwood, J. W., McKeown, N., Watson, G., Gibb, G., Hartke, P., Naous, J., Raghuraman, R., and Luo, J. 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, June 2007), pp. 160–161.

[16] Mallik, A., and Memik, G. 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.

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

[18] Shah, N., Plishker, W., Ravindran, K., and Keutzer, K. Np-click: A productive software development approach for network processors. *IEEE Micro 24*, 5 (Sept. 2004), 45–54.

[19] Spalink, T., Karlin, S., Peterson, L., and Gottlieb, Y. 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.

[20] Tennenhouse, D. L., and Wetherall, D. J. Towards an active network architecture. *ACM SIGCOMM Computer Communication Review 26*, 2 (Apr. 1996), 5–18.

[21] Turner, J. S. 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.

[22] Turner, J. S., Crowley, P., DeHart, J., Freestone, A., Heller, B., Kuhns, F., Kumar, S., Lockwood, J., Lu, J., Wilson, M., Wiseman, C., and Zar, D. 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.

[23] Wolf, T., Weng, N., and Tai, C.-H. Run-time support for multi-core packet processing systems. *IEEE Network 21*, 4 (July 2007), 29–37.