Runtime Resource Allocation in Multi-Core Packet Processing Systems

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

***Abstract*—Packet forwarding operations in network systems are often performed in software so that routers can be updated as new protocols and service features are developed. To meet the processing demands of high-performance networks, multi-processor systems-on-a-chip with dozens of cores are employed to provide raw processing power. Management of these processors and other system resources to achieve high forwarding rates is a key challenge. In particular, the allocation of processing workloads and the placement of data structures in memory have an enormous impact on system performance. Our work proposes a runtime system that manages these system resources. Much related work has proposed the use of cache memory hierarchies in packet processors. In this work, we show that our dynamic placement strategy can outperform a conventional cache memory and achieve up to 1.77 times higher hit rates for small memories, which are typically found in packet processing systems.**

***Index Terms*—network processor, memory, cache, runtime system**

I. Introduction

The data path of routers, where packets are forwarded from the input interface to the output interface, requires a number of packet processing steps. These packet processing steps involve basic protocol operations (e.g., standard IP forwarding [1]) as well as more advanced functionality (e.g., firewall [2], intrusion detection [3], network address translation [4]). Modern routers have a large number of configurable data path features ranging from accounting to load balancing and tunneling [5]. With recent efforts to develop a new network architecture for the next-generation Internet [6], even more functionality is being pushed into the data path of networks. Virtualized router systems use dynamically deployed, custom protocol stacks [7] to adapt to new requirements. Network services provide a mechanism for dynamically instantiate protocol processing features across a network [8].

To adapt to changes in the forwarding functionality, modern router designs have moved away from using application specific integrated circuits (ASIC), which cannot be reprogrammed once deployed. Instead, more general network processors (NP) based on embedded multi-processors systems-on-a-chip (MPSoC) have been proposed [9]. Such system exploit the inherent parallelism in network traffic (i.e., the lack of dependence between packets from different connections) to achieve throughput rates of several Gigabit per second using dozens of very simple processors. These multi-core packet processing engines are either implemented as systems specialized for networking (e.g., Intel IXP family [10], EZchip NP family [11]) or as general-purpose multi-processors that are adapted to networking functionality (e.g., Sun Niagara [12], MIT Raw [13]). As in many other computer systems, the memory system associated with these processors consists of a combination of memories ranging from fast but small (i.e., SRAM) to large but slow (i.e., DRAM).

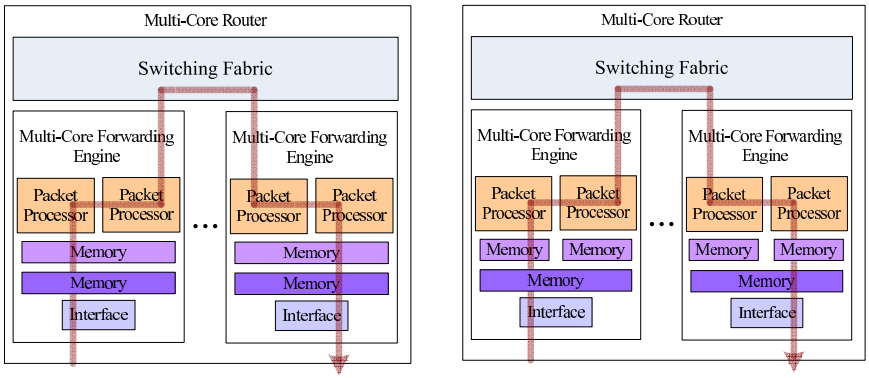
While multi-core packet processors provide the necessary computational performance, they also present a significant development and operational challenge. The parallelism of the system makes it difficult to program. Also, numerous components (e.g., processor cores, memory interfaces, hardware accelerators, etc.) need to interact smoothly to achieve maximum operational performance. Since many systems operate on the principle of a (software or hardware) pipeline, bottlenecks can cause considerable drops in performance. Therefore it is important to consider the following issues:

- Programming Abstractions: Packet processing applications need to be represented by a suitable programming abstractions to allow for effective exploitation of system-level and processor-level parallelism. In many cases, applications are represented by a directed graph of processing steps indicating order.
- Resource Allocation: Given the packet processing application, it is necessary to allocate system resources (e.g., processors, memory, etc.) to different processing task. A suitable resources allocation is crucial when aiming for high system performance.
- Runtime Adaptation: Resource allocation depends greatly on the workload demands put forth by the network traffic handled by the router system. Since network traffic changes during runtime, it is necessary to adapt resource allocation accordingly.

In this paper, we propose a solution to the problem of runtime resource allocation. In particular, we propose a resource allocation mechanism that considers both processing resources and memory resources. Prior and related work has considered these resources independently, but we argue that a runtime system that considers both in combination can perform better. The specific contributions of our work are:

- The design of a runtime management system that can

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



(a) Shared Memory Hierarchy (b) Dedicated and Shared Memories

Fig. 1. Multi-core packet processing system. architecture

profile processing and memory requirements of packet processing applications and perform dynamic resource allocation.

- A resource allocation algorithm for placing processing tasks on processors and for placing program data structures in memories.
- An evaluation of the proposed system on synthetic and realistic network traces that shows effective processor utilization and low memory access overheads. Compared to a system with static task placement and a cache memory hierarchy, our system achieves nearly optimal processor utilization and 1.77× higher first level memory hit rates.

The remainder of this paper is organized as follows. Section II discusses related work. Section III introduces system issues related to packet processing. The allocation algorithms are introduced in Section IV. Experimental results are presented in Section V. Section VI concludes this paper.

II. Related Work

Programming support for packet processing applications has been studied widely. Solutions range from low-level instruction set extensions [14], [15] to network-processor-specific language extensions [16] and high-level composition mechanisms for network operations [17]. One of the most general approaches is the Click programming model, which allows a composition of basic packet processing tasks as a queuing network [18]. Click has been extended for multi-processor support [19] and with network-processor specific functionality [20]. Our work is based on the Click abstraction that is slightly modified to permit task placement on different processors and to represent program data structures.

The task allocation problem on multi-core packet processing systems has been addressed in various forms. Static workload placement is well-understood [20], [21]. Runtime support for pipelined structures has been considered [22]. More general task distribution for dynamic workloads has been studied more recently [23]–[25]. The idea of using runtime profiling for more improved task placement has been described in our prior work [26], [27]. The main shortcoming of all these approaches is that they focus only on task allocation and do not further consider the placement of data structures in memory. Memories in packet processing systems have been considered from the perspective of design alternatives [28]. Also, dynamic allocation of data structures to memory has been attempted in general for parallel processors [29] and embedded systems [30], [31]. In our work, we consider a dynamic placement approach that is based on runtime profiling information. Combining task placement and program state allocation in packet processing system through a single runtime system is a novel approach presented in this paper.

III. System Overview

To provide context for the allocation algorithms presented in Section IV, we provide a brief overview about packet processing systems, throughput performance, and resource allocation issues.

A. Packet Processing Systems

A generic illustration of a router with multi-core packet processing engines is shown in Figure 1. Traffic enters the system through an interface on the router port. Then it is stored and processed by software running on one or more processor cores. Packets are then passed through the switching fabric to the output port where they may be processed and/or queued before exiting the router on the outgoing interface.

Practically all packet processing systems employ a hierarchy of memories (as shown in Figure 1). Small, low-latency memories (e.g., SRAM) provide fast access to frequently used data structures and larger, slower memories store less frequently accessed data. For example, the Intel IXP2400 system [10] uses three types of memories: Scratchpad SRAM for parameters and inter-processor communications, QDR SRAM for packet queue storage and lookup tables, and DDR DRAM for packet storage. Depending on system design, lower levels of the memory hierarchy (i.e., memories that are closer to the processor) may be shared (as shown in Figure 1(a)) or dedicated to a single processor (as shown in Figure 1(b)). For our work, we assume a shared memory hierarchy as shown in Figure 1(a).

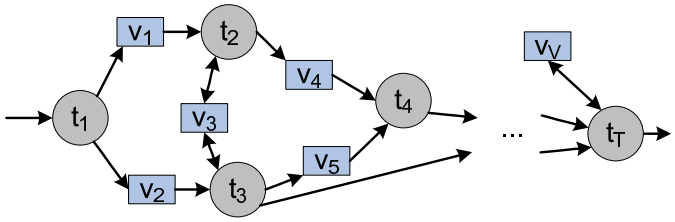


Fig. 2. Model of processing workload and program data structures.

B. Parallelism and Processing State

To exploit parallelism in packet processing workloads, packet processing tasks are distributed to different processor cores. Packet processing may be performed in a "run-to-completion" model, where a single processor core processes a packet in its entirety, or in a pipeline model, where packets are passed from core to core as different steps are completed. In many modern packet processors, a pipeline (with possibly multiple cores per stage) is employed. Therefore, it is important to consider how the processing tasks can be partitioned across processor cores.

One aspect of processing that needs to be considered carefully are program data structures. These data structures are used to maintain processing state and to store the packets that are being processed. Many modern packet processing functions use "stateful" processing (i.e., information that is carried between packets of the same flow) and thus data structures and their placement in the memory hierarchy of the system are becoming a more important topic.

C. Workload and Data Structure Representation

As discussed in Section II, there are a number of different abstractions used to represent packet processing applications. Our work focuses on allocation of processing tasks and program data structures and thus both components need to be represented suitably:

- Workload Partitioning: The granularity at which application workload is represented determines the basic mapping unit that an algorithm works on. In general, the higher the level of representation is, the coarser the mapping results are. Thus, it is important to consider at which level to represent such information. The finest level of representation, where individual processor instructions are considered, requires a very large amount of information and is very complex to use in later stages of the mapping process (e.g., large overhead for profiling). The coarsest level, where the application is seen as a single monolithic block, lacks details about the application and leads to trivial (and likely low performance) mapping results.

  In our work, we chose to represent applications at the level of basic functional blocks (i.e., "tasks") similar to Click modules [18]. Tasks are illustrated as round nodes in Figure 2. Sequences of arrows that connect tasks show the processing dependency among processing nodes.
- Program State Representation: In our work, we identify each data structure that is accessed by a task. This is illustrated as rectangular node in Figure 2. Based on read/write dependencies, arrows are unidirectional or bidirectional. Data structures can be shared by multiple tasks.

Using this abstraction of processing tasks and data structures, we propose allocation algorithms in Section IV.

D. System Performance

To understand the impact of resource allocation on system performance, we briefly discuss the basic performance model used in our work. Note that the model presented here is very simple. In prior work, we have developed a more detailed and more accurate performance model [32]. However, the complexities of the more advanced model may distract from the main point of this paper, which is resource allocation. Therefore, we use the simpler model, which shows the same performance tradeoffs, but may not achieve the same level of accuracy.

Typical performance metrics considered for packet processing systems are throughput, packet delay, power consumption, etc. For this work, we aim at maximizing system throughput, which is most important for many practical systems. On a multi-core packet processing system with shared memories, multiple cores execute instructions in parallel and generate requests to memory. Thus, the program execution time, t_{exec}, depends on the time spent for processing and the time spent for memory accesses. Since most packet processors use some form of a pipeline, the execution time is determined by the core that requires most processing time among the available N cores. Thus, we obtain the following equation:

$$t_{exec} = \max_{i=1...N} (t_{proc}(i)) + t_{mem}, \quad (1)$$

where $t_{proc}(i)$ is the time spent for processing by core i and t_{mem} is the time spent for memory accesses.

To increase system throughput, we need to decrease the total execution time, t_{exec}. There are two ways of achieving this goal:

1) Balance processing workload among processor cores: While the total amount of processing, $\sum_{i=1}^{N} t_{proc}(i)$, cannot be changed (unless network traffic changes), the worst case processing allocation, $\max_{i=1...N} (t_{proc}(i))$can be reduced. In the ideal case, all cores require the same processing time $(t_{proc}(1) = \cdots = t_{proc}(N))$.
2) Reducing memory access time: By reducing the amount of time spent for memory accesses, t_{mem}, the overall throughput can be achieved.

The algorithms presented in Section IV pursue exactly these two goals.

IV. Allocation Algorithms

In order to make a suitable allocation choice, it is necessary to know the demands of network traffic.

A. Runtime Profiling

As network traffic changes, the amount of processing associated with a task can change (since packets may take different paths through the workload graph illustrated in Figure 2). Therefore it is important to develop allocation mechanisms that can adapt at runtime. The general concepts of runtime adaptation are discussed in our prior work [26], [27]. The main idea is to obtain profiling information at runtime and to adapt accordingly. In our system, we profile two types of information: processing requirements and data structure accesses.

1) Profiling of Processing Demand: To characterize processing demand, the following information is collected at runtime:

- Task Service Time s_i: For each task t_i, we determine the service time s_i (measured, for example, in number of instructions executed per packet). Since this value may be different for each packet, we consider s_i as a sample from a random variable S_i. We assume the distribution of S_i matches the empirical observations of s_i.
- Task Utilization $u(t_i)$: Based on edge utilization (or through direct profiling), we can derive the utilization of a particular task t_i, which is denoted by $u(t_i)$.

Task service time reflects computational demands of tasks (i.e., expected service time $E[s_i]$), while task utilization represents how frequently tasks are used. We therefore define w_i as the amount of "work" that is imposed by task t_i:

$$w_i = u(t_i) \cdot E[S_i]. \tag{2}$$

Since "work" is measured in number of instructions per packet, the sum of all "work" on a processor yields $\sum_{i=1}^{N} t_{proc}(i)$.

2) Profiling of Data Structure Accesses: To obtain profiling information on memory accesses, we track size and utilization of data structures. We collect the following information about memory accesses:

- Data Structure Size p_i: For each data structure, we measure the size p_i (in number of bytes). For programs that do not allocate memory at runtime, this measurement could be done at offline to reduce runtime overhead.
- Data Structure Utilization $u(v_j)$: During the runtime of each processing tasks, we observe the amount of memory access (read and write) to each data structure from all tasks.

Using these profiling data, we can then run the allocation algorithms to determine resource allocation. As network traffic demands change the profiling data changes accordingly and the allocation process can be repeated. Using this process, the packet processing system can keep up with changes in the network.

B. Resource Allocation Problem

Assume a graph representing the packet processing workload is given (e.g., Figure 2) and profiling annotations are available as described above. Assume there are T task nodes, $t_1, \ldots, t_T$, and V data structures, $v_1, \ldots, v_V$, that are connected by directed edges $e_{i,j}$ with $i \in \{t_1, \ldots, t_T\}, j \in \{v_1, \ldots, v_V\}$ or $i \in \{v_1, \ldots, v_V\}, j \in \{t_1, \ldots, t_T\}$. (Bidirectional edges are represented by two directional edges in opposite direction.) Also assume that we represent a packet processing system by N processor cores and L layers of shared memory with capacities $C_1, \ldots, C_L$.

The resource allocation problem is to find a mapping m that assigns each of the T tasks to one of N processor cores and each of the V data structures to one of L memories: $m : \{t_1, \ldots, t_T\} \rightarrow [1, N], \{v_1, \ldots, v_V\} \rightarrow [1, L]$, such that the overall program execution time given by Equation 1 is minimized. This allocation needs to consider the constraint of resource limitations: $\forall j, 1 \leq j \leq L : |\sum_i \{v_i \times p_i | m(v_i) = j\}| \leq C_j$.

As discussed above, the goals of balancing processing time and minimizing memory access time can be considered independently from each other. Thus we first consider how to balance the mapping of processing tasks and then how to minimize memory access time.

C. Mapping of Processing Tasks

When mapping tasks to processors, we need to consider each task's service time. The problem of how to map tasks to processors such that the maximum of total all service times on processors is minimized is a load balancing problem. Unfortunately, this challenge is a "bin packing" problem and thus NP-hard. We therefore design a heuristic algorithm to achieve a good approximation of the optimal solution.

Algorithm 1 Task Mapping Algorithm.

Require: $w_1 \geq w_2 \geq \cdots \geq w_N$
1: $i \leftarrow 1$
2: $t_{proc}(1) \leftarrow 0, \ldots, t_{proc}(N) \leftarrow 0$
3: **while** $i \leq T$ **do**
4: $\quad k \leftarrow \text{argmin}_{j=1\ldots N}\,(W_j)$
5: $\quad m(t_i) \leftarrow k$
6: $\quad t_{proc}(k) \leftarrow t_{proc}(k) + w_i$
7: $\quad i \leftarrow i + 1$
8: **end while**
9: **return** m

Algorithm 1 uses a first fit decreasing strategy. Tasks are allocated in decreasing order of work w_i. A task is always allocated to the processor with least amount of work allocation (represented by $t_{proc}(i)$). It can be shown that the maximum workload allocation, $W = \max_{i=1\ldots N} t_{proc}(i)$, achieved by this algorithm's mapping m is no more than 1.5× that of the optimal mapping m^*. This proof follows the ideas in [33].

Proof: Assume an optimal task mapping m^*, where $W^* = \max_{j=1\ldots N} t_{proc}(j)$, and mapping m from Algorithm 1, where $W = \max_{j=1\ldots N} t_{proc}(j)$. Assume $T > N$ (otherwise $W^* = W$), then $W^* \geq \frac{\sum_i w_i}{N}$. Given that tasks are always mapped to the processor with minimum t_{proc}, the inequality $\min_{j=1\ldots N} t_{proc}(j) - w_T \leq \frac{\sum_j t_{proc}(j)}{N}$

holds when the last task t_T is to be mapped. Therefore, $\min_{j=1...N} t_{proc}(j) - w_T \leq \frac{\sum_j t_{proc}(j)}{N} \leq \frac{\sum_i w_i}{N} \leq W^*$. Since tasks are sorted by their workload in decreasing order before mapping and $|T| > N$, the inequality $w_T \leq w_{N+1} \leq \frac{W^*}{2}$ also holds for the last task t_T. Therefore, we have: $W = \min_j t_{proc}(j) - w_T + w_T \leq W^* + \frac{W^*}{2}$, which is $W \leq \frac{3 \times W^*}{2}$. ■

A key observation from the proof is that differences in task workload w_i are the main obstacle that prevents algorithm 1 from achieving a better approximation to optimal solution. To address this problem, we can use a technique called "task duplication," which duplicates processing intensive tasks (splitting the network traffic assigned to this task equally among duplicates). This approach is particularly useful in networking where there is little or no dependency between packets and thus duplication does not lead to coordination overhead. The detailed algorithm for task duplication is explained in [26].

Using these allocation strategies (either without or with duplication), we can achieve a balanced mapping of tasks to processing resources.

D. Data Structure Allocation

The memory allocation problem on a hierarchy of memories with different capacities and bandwidths can be formulated as a "0-1 knapsack" problem: Given are V data structures $v_1, \ldots, v_V$ and L memories with capacities $C_1, \ldots, C_L$. Each data structure v_i has a size of p_i and a utilization of $u(v_i)$. The goal is to find a mapping $m : \{v_1, \ldots, v_V\} \rightarrow [1, L]$, such that $\sum_i u(v_i)$ for fast memories is maximized while no memory capacity is exceeded ($\forall i, 1 \leq i \leq L : \sum_{j|m(v_j)=i} p_j \leq C_i$).

We design an algorithm using dynamic programming to solve this memory utilization problem:

To better utilize faster memories, we attempt to map data structures with higher utilization to those memories that have higher bandwidth. The algorithm requires memories to be sorted by their bandwidth. $A(x, y)$ is the maximum memory utilization that can be achieved with capacity less than or equal to y using data structures up to x. Boundary conditions $A(0, C_y) = 0$ and $A(v_x, 0) = 0$ hold for any set of capacities C_y and data structures v_x.

The mapping results provide an allocation of each data structure to a memory in the system. Therefore we can place these data structures accordingly. Implementation techniques for placing data structures at runtime have been described in [29]–[31].

V. Evaluation

Given allocation algorithms for processing tasks and data structures, we evaluate their effectiveness and compare them to existing approaches. We use PacketBench [34] to evaluate our algorithms through simulation and analysis. In our experimental configuration, we assume a packet processing system with eight processors and three layers of memories (similar to the Intel IXP2400 network processor). A collection of applications that are representative of various network services

Algorithm 2 Memory Mapping Algorithm.

```
1:  j ← 1
2:  while j ≤ L do
3:      n ← V
4:      A(n, C_j)
5:      V ← V ⋂ complement({v_i | m(v_i) = j})
6:      j ← j + 1
7:  end while
8:  return m
9:
10: function A(x, y)
11: if p_x ≤ y then
12:     if A(x − 1, y) ≤ u(v_x) + A(x − 1, y − p_x) then
13:         k ← u(v_x) + A(x − 1, y − p_x)
14:         m(v_x) ← j
15:     else
16:         k ← A(x − 1, y)
17:     end if
18: else
19:     k ← A(x − 1, y)
20: end if
21: return k
```

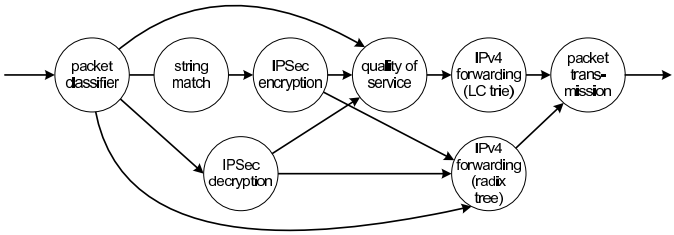


Fig. 3. Network services in experimental system.

are implemented on PacketBench for our evaluation (shown in Figure 3). The applications are further partitioned into 27 processing tasks which need to access around 4000 data structures.

The methodology of our experiments can be divided into four phases:

- Simulation: PacketBench is used to simulate the processing of a network packet trace. The simulation results from PacketBench are instruction traces that are passed to the profiling stage.
- Profiling: For each instruction in the instruction trace it is determined to which task it belongs. Memory access instructions and their target address and target data structure are identified. This information provides accurate profiling information for task service time s_i, task utilization $u(t_i)$, and data structure utilization $u(v_j)$.
- Mapping: Both processing task allocation and data structure allocation happen in this phase. Tasks are duplicated and mapped to cores and data structures are allocated to different layers of the memory system.
- Evaluation: After mapping, the quality of mapping for tasks and data structures is evaluated by determining

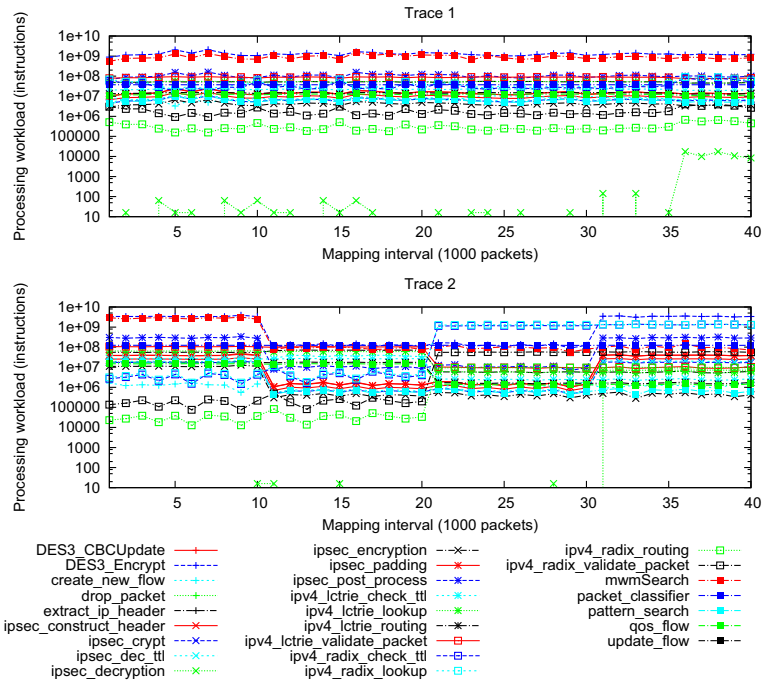


Fig. 4. Processing task workload.

processing balance and memory access time.

To evaluate our system with representative network loads, we have obtained packet traces from the Internet uplink of our institutional network. Trace 1 is an original network trace, which exhibits a low amount of dynamic variation in workload and access to data structures. Trace 2 is synthesized from four network traces obtained at different times, thus represents rapidly changing scenario.

To accommodate changes in network traffic, the profiling process takes place at intervals of 1000 packets. Mapping is revised after profiling information is available. Periodical revision of mapping averages out short bursts of network traffic, while maintaining adaptivity to the majority of network workloads.

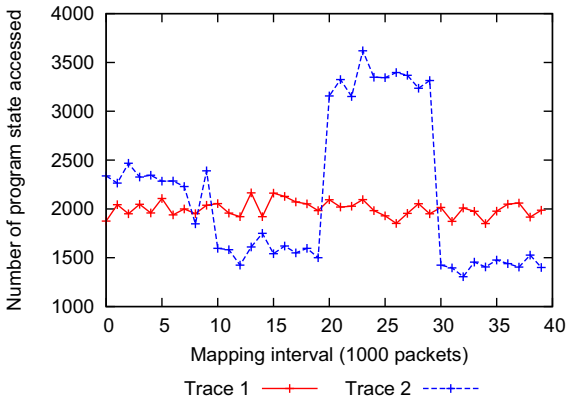


Fig. 5. Number of accessed data structures.

A. Workload Characteristics

Before discussing mapping performance, we show a brief analysis of the workload characteristics applied to the system. These results provide a context for the mapping results discussed below.

Task workload profiling results for both network traces are shown in Figure 4. A low level of variation in processing requirements can be observed in the workload for Trace 1. For Trace 2, the workload shows a high variation due to changes in the synthetic network trace every 10 intervals. Trace 2 illustrates a scenario where runtime adaptation is clearly necessary as the processing workload heavily depends on input traffic.

It can also be observed that the difference between individual workload can be very large (note the log scale y-axis in Figure 4). As observed above, the main obstacle that prevents Algorithm 1 from achieving better output is the difference in processing requirements for different tasks. Therefore task duplication is used in our system to get a well balanced workload on each processor.

Access to data structures change at runtime similar to how processing requirements change. The accessed subset of all data structures is dependent on input network traffic. Figure 5 shows the number of data structures that are accessed in each interval of Trace 1 and Trace 2.

The utilization of each data structure is also dependent on network traffic that exercises the system. Tasks require different amount of read and write operations based on the packets they are processing. Figure 6 shows the amount of bytes

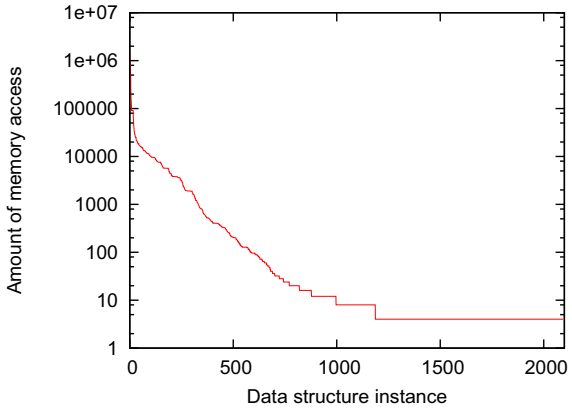


Fig. 6. Utilization of data structures in one interval.

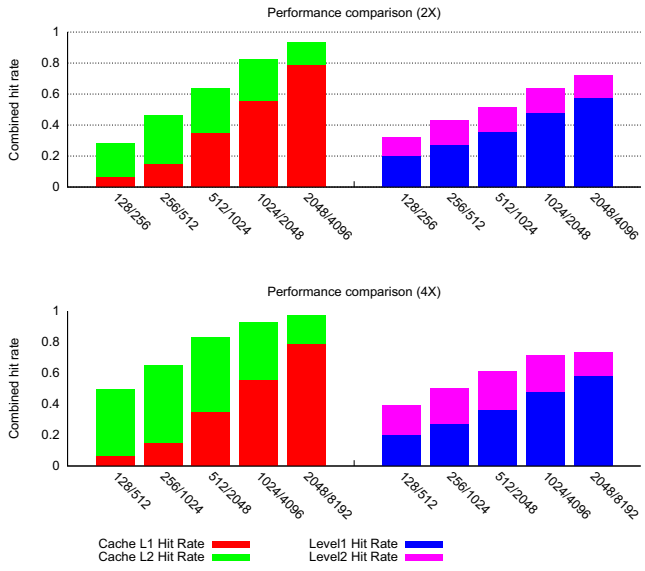


Fig. 7. Hit rate comparison for cached memories (left) and our allocation algorithm (right). X-axis shows memory size (level 1 / level 2) in bytes.

accessed from each of 2096 data structures in one interval. From this result, it is clear that differences in utilization of different data structures can be significant.

B. Comparison to Cache Memories

To put the performance of our data structure placement algorithm in context, we consider an alternative memory allocation approach based on a hierarchy of caches. We compare systems with equal amounts of memory. The cache system is configured with a block size of 64 bytes, an associativity of 2, and LRU replacement policy. To limit the design space, we assume that second level memories are a fixed factor larger than first level memories (either $2\times$ or $4\times$).

Figure 7 shows the hit rates for level 1 and level 2 caches on the left and the hit rates for our allocation algorithm on the right. We can observe that for small level 1 memory sizes,

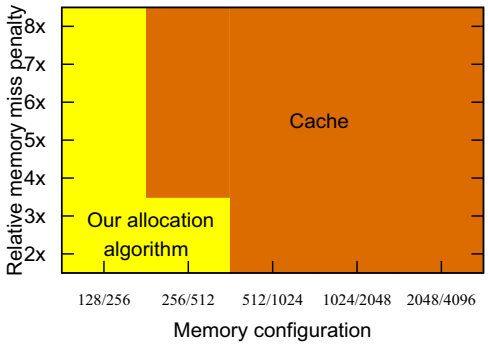


Fig. 8. Memory configuration space with indication of preferred data structure placement process.

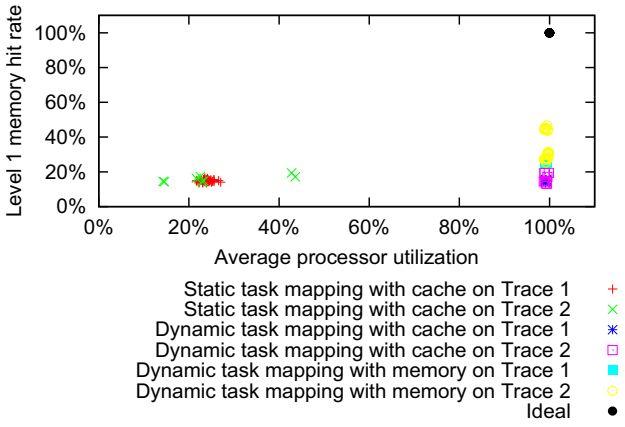


Fig. 9. Comparison of allocation quality.

as they are typical in network processors (e.g., Scratchpad memory in Intel IXP), our system achieves a higher hit rate in the first level memory. This increased hit rate has significant performance benefits since access to higher level memories are costly.

To illustrate for which memory configurations our system performs better, Figure 7 shows two regions with different shading. The y-axis shows the factor of miss penalties increase for accessing higher levels in memory hierarchy. Our system performs better for small memory sizes and low relative miss penalties.

C. Mapping Quality

For overall performance evaluation of the allocation algorithms, we show the first level memory hit rate and the average processor utilization in Figure 9 (256 bytes of first level memory). The average processor utilization depends on the balance achieved by the processing task mapping. In the ideal case, it is 100%. The first level hit rate is also 100% in the ideal case.

Static application mapping is a baseline, where tasks are assigned to different processors independently of network traf-

fic. This mapping is similar to conventional task management on programmable packet processing systems. As expected the utilization is low due to large differences in processing task requirements. Also, the hit rate into first level memory is low. Such a system does not perform well. In comparison, dynamic task mapping achieves nearly 100% utilization. Using our memory allocation algorithm, a much higher hit rate (up to around 45%) can be achieved. In comparison, cached memories achieve hit rates in the order of 15–30%. On average, our system shows an improvement of first level memory hit rate of 1.77× over cache memories.

VI. Conclusion

Runtime management of processing and memory resources on multi-core packet processing systems is an important problem. We present two algorithms that allocate processing tasks to cores and program data structures to memories. Our system achieves nearly optimal system utilization and up to 1.77× higher hit rates into first level memory than cached memories. In particular, our system performs better than caches in very small memory configurations as they are encountered on practical packet processing systems.

References

[1] F. Baker, "Requirements for IP version 4 routers," Network Working Group, RFC 1812, June 1995.

[2] J. C. Mogul, "Simple and flexible datagram access controls for UNIX-based gateways," in *USENIX Conference Proceedings*, Baltimore, MD, June 1989, pp. 203–221.

[3] *The Open Source Network Intrusion Detection System*, Snort, 2004, http://www.snort.org.

[4] K. B. Egevang and P. Francis, "The IP network address translator (NAT)," Network Working Group, RFC 1631, May 1994.

[5] 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.

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

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

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

[10] *Intel Second Generation Network Processor*, Intel Corporation, 2005, http://www.intel.com/design/network/products/npfamily/.

[11] *NP-3 – 30-Gigabit Network Processor with Integrated Traffic Management*, EZchip Technologies Ltd., Yokneam, Israel, May 2007, http://www.ezchip.com/.

[12] 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.

[13] S. Amarasinghe, G. Chuvpilo, and D. Wentzlaff, "Gigabit IP routing on Raw," in *Proc. of First Network Processor Workshop (NP-1) in conjunction with Eighth IEEE International Symposium on High Performance Computer Architecture (HPCA-8)*, Cambridge, MA, Feb. 2002, pp. 2–9.

[14] M. Grunewald, D. K. Le, U. Kastens, J.-C. Niemann, M. Porrmann, U. Ruckert, A. Slowik, and M. Thies, "Network application driven instruction set extensions for embedded processing clusters," in *Proc. of the International Conference on Parallel Computing in Electrical Engineering (PARELEC)*, Dresden, Germany, Sept. 2004, pp. 209–214.

[15] B. Li and R. Gupta, "Bit section instruction set extension of ARM for embedded applications," in *Proc. of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, Grenoble, France, Oct. 2002, pp. 69–78.

[16] L. George and M. Blume, "Taming the IXP network processor," *ACM SIGPLAN Notices*, vol. 38, no. 5, pp. 26–37, May 2003.

[17] A. T. Campbell, S. T. Chou, M. E. Kounavis, V. D. Stachtos, and J. Vicente, "NetBind: a binding tool for constructing data paths in network processor-based routers," in *Proc. of the Fifth IEEE Conference on Open Architectures and Network Programming (OPENARCH)*, New York, NY, June 2002, pp. 37–44.

[18] 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.

[19] 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, June 2001, pp. 333–346.

[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, Sept. 2004.

[21] 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.

[22] M. K. Chen, X. F. Li, R. Lian, J. H. Lin, L. Liu, T. Liu, and R. Ju, "Shangri-la: achieving high performance from compiled network applications while enabling ease of programming," in *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, Chicago, IL, June 2005, pp. 224–236.

[23] 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.

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

[25] X. Huang and T. Wolf, "Evaluating dynamic task mapping in network processor runtime systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 8, pp. 1086–1098, Aug. 2008.

[26] 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.

[27] ——, "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.

[28] J. Mudigonda, H. M. Vin, and R. Yavatkar, "Overcoming the memory wall in packet processing: Hammers or ladders?" in *Proc. of ACM/IEEE Symposium on Architectures for Networking and Communication Systems (ANCS)*, Princeton, NJ, Oct. 2005, pp. 1–10.

[29] C. A. Moritz, M. Frank, W. Lee, S. Amarasinghe, and A. Agarwal, "Hot Pages: Software caching for Raw microprocessors," Massachusetts Institute of Technology, Cambridge, MA, Technical Memo LCS-TM-599, Aug. 1999.

[30] S. Udayakumaran and R. Barua, "Compiler-decided dynamic memory allocation for scratch-pad based embedded systems," in *Proc. of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, San Jose, CA, Oct. 2003, pp. 276–286.

[31] N. Nguyen, A. Dominguez, and R. Barua, "Memory allocation for embedded systems with a compile-time-unknown scratch-pad size," in *Proc. of the 2005 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, San Francisco, CA, Sept. 2005, pp. 115–125.

[32] N. Weng and T. Wolf, "Analytic modeling of network processors for parallel workload mapping," *ACM Transactions on Embedded Computing Systems*, vol. 8, no. 3, pp. 1–29, Apr. 2009.

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

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