Distributed Runtime Load-Balancing for Software Routers on Homogeneous Many-Core Processors

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

ABSTRACT

With the advent of diversifie network services and programmability deployed in the network infrastructure, the functionality of the data path in network systems has moved from "store-and-forward" toward "store-process-forward." However, the processing performance of many contemporary software routers does not scale with the increasing number of processor cores that are integrated on a chip due to software bottlenecks. To tackle one aspect of this problem, we propose a distributed algorithm that can load-balance packet processing workloads on a modern many-core architecture. The algorithm exploits parallelism and achieves load balancing by distributing processing task across different local regions of the chi. Workload distribution at chip level can be achieved with an $O(n \log n)$ time complexity and thus can scale to large configurations

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

Routers with packet processing functionalities implemented in hardware have been deployed at the core of networks due to their ability to forward packets at much higher rates than software router solutions. However, the data plane of today's Internet needs to be extensible to adapt to user demands for services such as security, monitoring, and content transcoding. In contrast to hardware approaches, software routers provide the necessary flexibility that enables quick development of network functionalities and dynamic deployment. Such software routers serve as basis for many new network applications (e.g. network virtualization).

With development of modern many-core architectures in the last decade, an increasing number of processor cores integrated on a single chip give software routers potentially high performance. To translate the massive raw processing power of these systems into high packet processing performance in software routers, three key challenges need to be addressed:

- Processing workload partitioning: With dozens to thousands of processing cores, network applications written with traditional programming abstractions need to be adapted to provide enough schedulable tasks to execute in parallel on a large number of cores.
- Utilization of hardware resources: Load balancing is difficult to achieve on many-core architectures. Even without considering utilization of the core-to-core interconnect, task scheduling is equivalent to the Bin Packing problem, which is NP-hard.
- Adaptation to network traffic: Network traffic is bursty in nature. Static resource allocation cannot support the full range of possible workload scenarios. The ability to dynamically adapt to drastic changes in processing demand is crucial for high-performance software routers.

In this paper, we look at these challenges and present our solution to the problem of adaptive allocation of hardware resources. Specifically, our contributions in this paper are:

- A design of runtime mechanism to solve the load-balancing problem on a homogeneous many-core architecture.
- A distributed algorithm that scales with the number of cores integrated in the system and schedules processing tasks incrementally with changes in network traffic.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profi or commercial advantage and that copies bear this notice and the full citation on the firs page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specifi permission and/or a fee.
ACM PRESTO 2010, November 30, 2010, Philadelphia, USA.
Copyright 2010 ACM 978-1-4503-0467-2/10/11 ...$10.00.

Section 2 presents background and related work. We introduce the targeted many-core architecture used in our software routers in Section 3. Workload partitioning and programming abstraction for software routers are presented in Section 4. Section 5 provides a detailed discussion of our distributed scheduling algorithm. We discuss future work and conclude this paper in Section 6.

2. RELATED WORK

Attempts to design novel many-core architectures that scale with increasing VLSI gate density have been made in several areas in last decade. The MIT Raw project [1], the Tilera Tile64 [2], and the recent Intel single-chip cloud computer (SCC) [3] aim at general computing. In the networking domain, network processors [4, 5] have also shifted toward many-core architectures.

Software routers can be developed on various platforms ranging from general computing systems such as workstation or server [6, 7] to specialized hardware such as network processors [8] and graphics processors [9]. With an increasing number of cores integrated on-chip, the traditional run-to-completion model for packet processing becomes inefficient for high data rates. To tackle this problem, recent software routers based on general computing platforms already move toward more flexible schemes for offloading processing [10].

A simple and concise method to partition network processing applications and create large number of schedulable elements has been attempted by Click [11]. Load-balancing scheduling of Click-based software router has been developed on symmetric multiprocessors (SMP) architectures [12, 13] and network processors [14].

Achieving high utilization of massively parallel processing resources has been studied in the context of distributed operating systems for traditional distributed systems [15], many-core processors, and cloud computing [16]. Our work on many-core architectures differs from these results because we specifically focus on network processing workloads.

3. MANY-CORE ARCHITECTURE

Traditional SMP multi-core architectures such as the Intel Core Duo [17] have been extended from single-core system. On such architectures, cores have individual low-level caches to exploit locality in program execution and shared high level cache to maintain global cache coherence. Large amounts of data are exchanged among cores and the execution of synchronization primitives are carried on the shared memory bus. The competition for access to the memory bus therefore becomes the bottleneck of the architecture and has limited such systems from scaling to more than a dozen of cores. This limitation, in turn, has led to low performance of network processing [18].

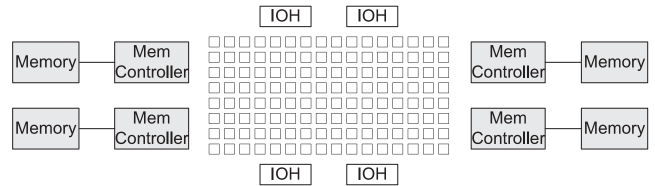


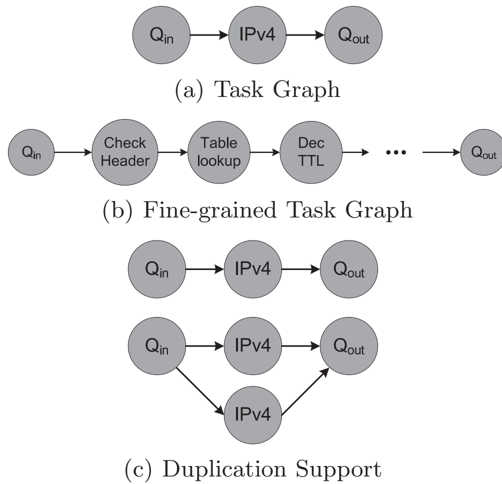
Figure 1: Homogeneous Many-Core Architecture

To tackle this scalability issue, architectures of current many-core processors, such as Tilera Tile64 and Intel SCC, have been designed with dozens to hundreds of cores interconnected by a mesh network, as shown in Figure 1. Processor cores are divided into regions with each region having its own I/O hub and local memory (NUMA). Memory and I/O access to local resources is significantly faster than to remote resources on such architectures, but global cache coherence is not supported in hardware. These innovations in hardware design present significant challenges to both software development and resource management. Key problems are how to utilize the large number of cores and how to balance the processing workload across local hardware resources without creating performance bottlenecks. These questions have become a challenging and urgent issue in both research and development.

4. SOFTWARE ROUTER SYSTEM

Raw processing power and high interconnect bandwidth make homogeneous many-core architecture an ideal platform for software routers to perform packet-oriented processing. However, current software router systems do not scale well on such architecture for three reasons:

- Cache inefficiency: Typical software routers run the network stack in a monolithic kernel, which requires more memory than can fit into the cache of most off-the-shelf processors. This leads to cache miss penalties that have significant negative impact on system throughput. Additional packet processing steps and large amounts of input traffic may further contribute to high cache miss rates.
- High synchronization cost: The run-to-completion model for packet processing (i.e., all processing for a single packet stays in one processor core) works well for SMP architectures with a dozen or less cores. However, the inherent problem of synchronization (e.g., lock mechanism) for run-to-completion model severely limits the scalability of software system [16].
- Low single-core performance: The performance increase of individual cores does not match that



(a) Task Graph

(b) Fine-grained Task Graph

(c) Duplication Support

Figure 2: Workload Representation for Software Router

of network devices. The performance of individual processor cores grows slowly and most performance increases on processor systems are due to larger numbers of parallel cores. With the advent of 40 Gbps network equipment, packets from one input device can easily saturate an entire core. This trend implies that packet processing systems will continue to move towards many-core architectures, for which the run-to-completion model is unsuitable.

Micro-kernel operating system with the ability to flexibly distribute packet processing workload among cores is therefore an obvious choice for the next-generation software router infrastructure.

To achieve flexible workload offloading, the traditional tightly-coupled run-to-completion programming model needs to be broken into loosely-coupled processing steps. An example of such an approach is illustrated in Figure 2(a), where a "task graph" of connected processing steps is shown. Each node in task graph represents part of the overall processing workload and can be scheduled individually on any existing core. Distribution of such workload on many-core systems can be explored in two dimensions:

- Exploiting parallelism between task graph nodes: Fine-grained workload partition yields more nodes that can potentially be scheduled on more cores. Figure 2(b) shows an example of partitioning IPv4 forwarding workload. Steps for TTL field and checksum checking, forwarding table lookup, TTL update and etc. are identified separately as schedulable nodes similar to elements in Click [11].

- Exploiting parallelism within a task graph node: Multiple instances of a node can be scheduled on different cores at the same time to split the overall workload of original node [19]. For example, all processing in IP forwarding can be done in parallel since duplicated nodes can work on different packets and no synchronization is required. The ability of scheduling duplicated nodes is essential to enabling massive parallelism in many-core architecture. Some nodes, such as the device input queue Q_{in} and output queue Q_{out} in Figure 2(c), can have only a fixed number of duplicated instances as each instance handles a separate queue in physical device. Other nodes, such as IP forwarding, can be duplicated arbitrarily many times to let many cores in the system run an instance for maximum parallelism.

The first level of parallelism is up to software developers. In this paper, we present a runtime mechanism that is designed to schedule multiple task graphs and multiple instances of tasks to exploit the full parallelism of a many-core architecture.

5. DISTRIBUTED RUNTIME SCHEDULING

We designed a distributed run-time system to manage processing resource dynamically on a many-core system. Each core runs a micro-kernel operating system which handles message construction and transmission, tasks instantiation, and local time-sharing scheduling (e.g., using [20]). The target many-core architecture is assumed to have common constraints such as bidirectional interconnects, reliable message delivery, queuing policy, total connectivity, and unique identifiers for each core.

The main idea of our approach is to identify the cores that are overloaded (and thus present performance bottlenecks). Using the offloading algorithm described below, we reassign processing tasks from these highly-loaded cores to achieve a more balanced workload distribution. To minimize the overhead for moving processing tasks over long distances, we give preference to offloading to immediate neighbors. However, the system also supports offloading across clusters or the entire many-core system. Providing these offloading techniques ensures that workload hot spots get dynamically dispersed over the entire chip.

In this section, we first discuss how resource consumption is monitored and profiled in the system. Then, we introduce task offloading in mesh networks with a few cores. At the end of the section, we extend our distributed algorithm to mesh network of arbitrary size.

5.1 Resource Consumption Profiling

For each task running on a processor core, the local operating system associates an input queue and output queue with the task. As the system is in charge

of packet exchange with neighboring cores and task scheduling, it is capable of knowing the I/O bandwidth to neighboring cores and the processing capacity allocated to each local task. With this profiling information, the system can make decisions about the status of a local task, such as whether it is bounded by I/O (i.e., small and fixed queue length) or processing (i.e., large and increasing queue length). The goal of task offloading in such a many-core system is to distribute processing-bounded tasks to under-utilized cores, for which there is sufficient interconnect bandwidth between offloading core and offloaded core.

5.2 Local Task Offloading

We have designed a heuristic algorithm (Algorithm 1) for task offloading between neighboring cores (`bfo_offload()`). Whenever the local system detects a processing-bounded task, it first notifies all its neighbors $N(PID)$ to finish their on-going offloading process with `lock_neighbor()`. This step ensures that the initiator node can get accurate utilization information from neighbors. It is important to note that locking does not interrupt tasks that are already running on neighbors. The algorithm then selects neighbors whose remaining bidirectional interconnect bandwidth B_i can at least match input/output bandwidth $B(task)$ of the to-be-offloaded task. If there exists such a neighbor, the algorithm further checks its unutilized processing capacity $C_{total} - C_N$, and offload the task only when the neighbor have more available processing capacity than the local processor.

System performance can also benefit from offloading a duplicated task instance to under-utilized neighbors. In this case, function `bfo_offload_dup()` lets the local system select the most under-utilized neighbor with unused bidirectional bandwidth and instruct this neighbor to instantiate a duplicated task.

This algorithm accomplishes best-effort task offloading from the local processor. When a task can be duplicated, the duplicating version of the algorithm is used by the system; otherwise the normal version is executed. However, Algorithm 1 can only handle task offloading among immediate neighbors. In mesh networks with dozens of cores or more it is necessary to enable task offloading between remote cores.

5.3 Task Offloading in Cluster

Figure 3 shows an example of a cluster. We use a spanning tree (grey lines) to exchange utilization information to identify the node within the cluster that is least loaded (and thus a candidate to receive offloaded tasks). We use $I_{i,util}$ to represent this utilization information for core i. On the control tree, the election of the candidate node follows the "saturation" technique described in [21], which has three phases:

Algorithm 1 Breadth-First Offloading.

1: **function** bfo_offload()
2: $result \leftarrow$ FAILURE
3: $N_{unvisited} \leftarrow N(PID)$
4: lock_neighbor($N_{unvisited}$)
5: **for** $i = |N_{unvisited}|$ to 1 **do**
6:     **if** $B_i < B(task)$ **then**
7:         $N_{unvisited} \leftarrow N_{unvisited} - N_i$
8:     **end if**
9: **end for**
10: **if** $|N_{unvisited}| > 0$ **then**
11:     $N \leftarrow \min_i(U(N_{unvisited}))$
12:     **if** $C_{total} - C_N > C_{task}$ **then**
13:         $result \leftarrow$ offload($task, N$)
14:     **end if**
15: **end if**
16: unlock_neighbor($N(PID)$)
17: **return** $result$
18:
19: **function** bfo_offload_dup()
20: $result \leftarrow$ FAILURE
21: $N_{unvisited} \leftarrow N(PID)$
22: lock_neighbor($N_{unvisited}$)
23: **for** $i = |N_{unvisited}|$ to 1 **do**
24:     $N \leftarrow \min_i(U(N_{unvisited}))$
25:     **if** $B_N > 0$ **then**
26:         $result \leftarrow$ offload($task, N$)
27:         break
28:     **else**
29:         $N_{unvisited} \leftarrow N_{unvisited} - N_i$
30:     **end if**
31: **end for**
32: unlock_neighbor($N(PID)$)
33: **return** $result$
34:
35: **function** lock_neighbor(N)
36: **for** $i = |N|$ to 1 **do**
37:     send M_{lock} to N_i
38:     wait until receive $M_{lockack}$
39: **end for**
40:
41: **function** unlock_neighbor(N)
42: **for** $i = |N|$ to 1 **do**
43:     send M_{unlock} to N_i
44:     wait until receive $M_{unlockack}$
45: **end for**
46:
47: **function** offload($task$,N)
48: send $M_{offload}$ to N
49: wait until receive $M_{offloadack}$
50: $result \leftarrow RESULT(M_{offloadack})$
51: **return** $result$
52:

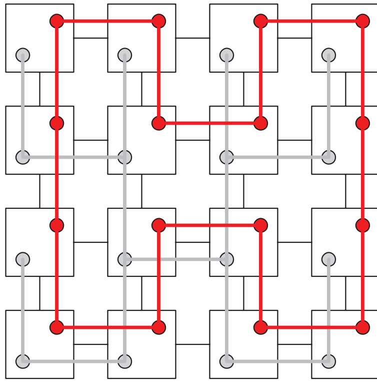


Figure 3: Control Tree and Token Ring

1. Activation: Whenever the current candidate starts to run a new task, it broadcasts a "re-election" message on the control tree.

2. Saturation: Each leaf node i receives "re-election," collects its $I_{i,util}$, and sends it to connected internal node; Any internal node j, after receiving "re-election" and collecting its own $I_{j,util}$ waits until receiving all but one neighbor's utilization information. Then it compares all available information including its own, selects the node k with minimum utilization, and sends $I_{k,util}$ to the only neighbor in control tree that it has not yet received information from. At the end of this phase, exactly two adjacent nodes receive information from all their neighbors. After these two exchange minimum utilization information on their part of the tree, they can agree on which core in the control tree has minimum utilization information, and this core is elected as candidate. In case multiple candidates exist, the one with largest identifier is elected.

3. Resolution: In this phase, the candidate's identifier and utilization is broadcasted on control tree such that each node knows to which core it should offload task to.

Message complexity of candidate election is $3n - 4$, where n is the number of cores in a cluster. Re-election only happens when a new task is offloaded to the candidate.

To prevent message flooding of simultaneous offloading requests from multiple cores, we use a token ring (red lines) that includes all cores in a cluster. A token is a message passed on the ring in a fixed direction. The core with the token checks whether it is feasible to offload a local task graph to candidate, and performs offloading if necessary, then pass the token on to next core. The use of token ring therefore enables offload initiator selection in a round-robin manner.

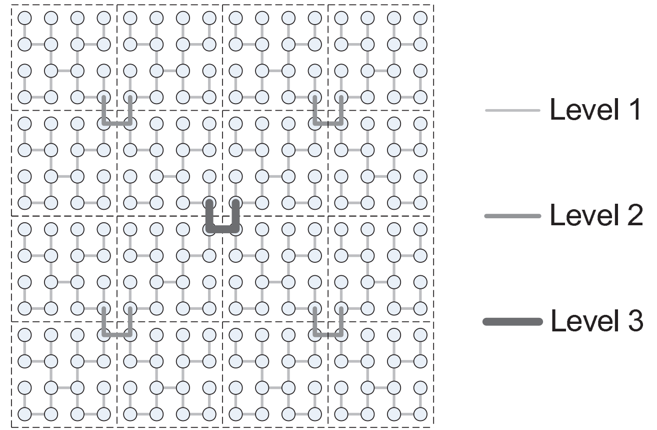


Figure 4: Multi-level Control Tree

Our candidate selection algorithm can be extended to select more than one candidate node in each run and to allow multiple tokens for offloading. These changes lead to more (parallel) offloading within a cluster.

5.4 Global Task Offload

The scalability of the control tree and token ring for clusters discussed above is limited and may not be suitable for clusters of large size. To tackle this problem, we extend the control tree to multiple levels. Figure 4 shows a three-level spanning tree that controls 256 cores.

The first-level spanning tree manages candidate selection in clusters. Higher-level trees are constructed from up to four lower-level trees. Whenever an offload request cannot be resolved on a ith level, this request will forwarded to $i + 1$st level. This action triggers candidate selection on the $i + 1$st level. In the limit, all levels are checked for suitable offloading candidates.

5.5 Scheduling Summary

Our scheduling mechanism manages processing and interconnect resources in many-core architecture in a divide-and-conquer manner: cores are divided into multiple levels of regions with the top-most region covering all nodes and leaf regions (clusters) containing fixed number of cores. Scheduling (i.e., offloading) attempts at any level are first made in the local region. Only when local region does not have sufficient resources (i.e., offloading fails due to the lack of a under-utilized core), the scheduling attempt goes up one level. Regions of the same level can process in parallel without the need to synchronize unless the higher-level region decides to make a cross-region offloading attempt. This approach ensures that the many-core system can process packets effectively and with only a small amount of coordination between cores.

6. CONCLUSIONS

Many-core architectures present a fundamental challenges to software router systems with respect to hardware resource utilization. As systems with thousands of cores on a single chip emerge over the coming decade, it is crucial to provide the mechanisms to translate this raw processing power into high packet processing performance. We present a system for managing processing resources that allows balanced distribution of task across the entire chip. Our distributed task offloading algorithm is capable of making local offloading decisions in parallel and chip-level offloading decisions within $O(n \log n)$ time complexity.

As ongoing research, we are planning to develop an implementation of the runtime mechanism on a mesh-based network processor prototype. Once this mechanism exists on a practical system, we can evaluate its performance and compare it to other techniques. In addition, we hope to show that the presented approach provides an ideal mechanism for dispersion of workload across a many-core architecture.

Overall, we believe that understanding and effectively utilizing many-core processor systems is essential for future high-performance network systems. The presented work provides an initial step towards this goal.

Acknowledgements

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

7. REFERENCES

[1] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The raw microprocessor: A computational fabric for software circuits and general-purpose programs," *IEEE Micro*, vol. 22, no. 2, pp. 25–35, 2002.

[2] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook, "Tile64 - processor: A 64-core soc with mesh interconnect," feb. 2008, pp. 88 –598.

[3] T. Mattson, R. Van der Wijingaart, M. Riepen, T. Lehnig, P. Brett, W. Hass, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe, "The 48-core scc processor: The programmers view," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, New Orleans, LA, USA, Nov. 2010.

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

[5] Q. Wu and T. Wolf, "Design of a network service processing platform for data path customization," in *Proc. of The Second ACM SIGCOMM Workshop on Programmable Routers for Extensible Service of TOmorrow (PRESTO)*, Barcelona, Spain, Aug. 2009.

[6] K. Argyraki, S. Baset, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, E. Kohler, M. Manesh, S. Nedevschi, and S. Ratnasamy, "Can software routers scale?" in *PRESTO '08: Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow.* New York, NY, USA: ACM, 2008, pp. 21–26.

[7] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "Routebricks: exploiting parallelism to scale software routers," in *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles.* New York, NY, USA: ACM, 2009, pp. 15–28.

[8] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb, "Building a robust software-based router using network processors," in *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles.* New York, NY, USA: ACM, 2001, pp. 216–229.

[9] S. Han, K. Jang, K. Park, and S. Moon, "Packetshader: a gpu-accelerated software router," in *SIGCOMM '10: Proceedings of the ACM SIGCOMM 2010 conference on Data communication*, Newdelhi, India, Sep. 2010.

[10] T. Herbert, "Receive packet steering: A software solution to scaling the network receive path," in *Linux Plumbers Conference*, Portland, OR, USA, Sep. 2009.

[11] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, 2000.

[12] B. Chen and R. Morris, "Flexible control of parallelism in a multiprocessor pc router," in *Proceedings of the General Track: 2002 USENIX Annual Technical Conference.* Berkeley, CA, USA: USENIX Association, 2001, pp. 333–346.

[13] Q. Wu and T. Wolf, "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.

[14] 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, 2004.

[15] D. S. Milojíčić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process migration," *ACM Comput. Surv.*, vol. 32, no. 3, pp. 241–299, 2000.

[16] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos): the case for a scalable operating system for multicores," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 2, pp. 76–85, 2009.

[17] S. Gochman, A. Mendelson, A. Naveh, and E. Rotem, "Introduction to intel core duo processor architecture," *Intel Technology Journal*, vol. 10, no. 2, 2006.

[18] N. Egi, M. Dobrescu, J. Du, K. Argyraki, B.-G. C. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, L. Mathy, and S. Ratnasamy, "Understanding the packet processing capabilities of multi-core servers," 2009. [Online]. Available: `http://infoscience.epfl.ch/record/134539`

[19] 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.

[20] Q. Wu, S. Shanbhag, and T. Wolf, "Fair multithreading on packet processors for scalable network virtualization," in *Proc. of ACM/IEEE Symposium on Architectures for Networking and Communication Systems (ANCS)*, San Diego, CA, Oct. 2010.

[21] N. Santoro, *Design and Analysis of Distributed Algorithms (Wiley Series on Parallel and Distributed Computing).* Wiley-Interscience, 2006.