Controlling Parallelism in a Multicore Software Router

Mihai Dobrescu, Katerina Argyraki
EPFL, Switzerland

Gianluca Iannaccone, Maziar Manesh,
Sylvia Ratnasamy
Intel Research Labs, Berkeley

ABSTRACT

Software routers promise to enable the fast deployment of new, sophisticated kinds of packet processing without the need to buy and deploy expensive new equipment. The challenge is offering such programmability while at the same time achieving a competitive level of performance. Recent work has demonstrated that software routers are capable of high performance, but only for conventional, simple workloads (like packet forwarding and IP routing) and, even that, after careful manual calibration. In contrast, we are interested in achieving high performance in the context of a software router running multiple sophisticated packet-processing applications. In particular: first, we identify the main factors that affect packet-processing performance on a modern multicore general-purpose server—cache misses, cache contention, load-balancing across processing cores; then, we formulate an optimization problem that takes as input a particular server architecture and a packet processing flow, and determines how to parallelize the router's functionality across the available cores so as to maximize its throughput.

1. INTRODUCTION

Software routers—network equipment, in which all packet processing is implemented in software, running on general-purpose processors—could be an attractive alternative to the traditional hardware routers. They could enable the development of truly programmable networks, where changing the network's functionality would be equivalent to performing a software upgrade. In other words, they could make it feasible to try out new kinds of packet processing without the need to buy and deploy expensive new equipment.

The Achilles' heel of software routers has traditionally been performance—a few years ago, when hardware routers were reaching aggregate throughput of Tbps [1], software routers had trouble scaling beyond the 1–5Gbps range [3]. Yet multicore technology is changing the scene: within the

last year, we have seen commercial network appliances running on general-purpose processors, capable of handling tens of Gbps [4]; on the research front, we have seen a filtering platform running on a general-purpose server, capable of performing multi-dimensional packet classification at 15Gbps [9], a parallel software router architecture, capable of scaling to hundreds of Gbps [6] and, more recently, an optimized I/O software architecture that significantly improves the packet-processing capability of one server and offers forwarding rates of up to 10Gbps per core[7].

These recent advances have demonstrated that software routers are capable of high performance *provided they are carefully programmed*. For instance, in the context of the RouteBricks prototype, a general-purpose server equipped with two quad-core Nehalem processors was able to perform IPv4 routing at 24Gbps [6]. This performance, however, was achieved only after tedious manual tuning. Moreover, all packets were subjected to the same kind of packet processing (IP routing) and all necessary data structures (the forwarding table) fit in the cache. A minor deviation from this careful setup could result in a significant performance drop.

Yet the allure of software routers is that they could enable the network to evolve *beyond* conventional IP routing. To prove this, it is not enough to demonstrate that a software router can achieve good performance in a very particular context; we need to demonstrate that a software router can achieve high performance for a range of sophisticated packet processing applications without loosing programmability.

This paper is a first attempt at determining how to simultaneously achieve both high performance and ease of programmability in a software router. For this, a high-level goal is to develop a compiler that automates the process of parallelizing a packet-processing flow. Hence, we ask: given a particular multicore server architecture and a particular set of packet-processing applications, how do we parallelize router functionality across the available cores so as to maximize throughput? and what do we need to know about the server architecture and each application in order to perform such an optimization? In some sense, our work builds on early work by Chen and Morris on SMP Click [5] that sought to extend Click for multi-processor architectures. They evaluated two approaches to scheduling a Click program across multiple cores: (1) the "dynamic" approach, where processing elements are assigned to cores in a manner that seeks to balance

CPU utilization across cores, without taking into account the overheads that result from partitioning the processing of a packet across different processors (e.g. cache misses); (2) the "static" approach, which relies on a static assignment of elements to cores—this assignment is determined manually by an "ambitious" programmer and is fixed for the duration of the program's execution. Their results suggest that the static approach typically outperforms the dynamic one. The goal of our work is to automate the manual assignment decision made by the ambitious programmer of SMP Click and, subsequently, to allow this scheduling decision to be made in a dynamic manner at runtime.

The rest of the paper is organized as follows. In Section 2, we describe what are the general approaches for exposing parallelism in a multicore software router. In Section 3, we describe the overheads incurred by each approach and formulate an optimization problem that determines which approach is better for a particular server architecture and a given workload. We outline the limitations of our model and our plan toward addressing them in Section 4, and conclude in Section 5.

2. SETUP

At a high level, our goal is: given a packet-processing application, we want a method that automates the parallelization of the application to maximize performance. This question falls, of course, under the general question of whether/how we can parallelize code to exploit multicore hardware—a hard problem and the subject of much ongoing research on multicore systems. For tractability, we narrow our exploration to a specific data-flow programming model and consider first the question of how we parallelize a single data-flow under several simplifying assumptions that we elaborate on as introduced in the text. While we believe our approach would generalize to multiple data-flows, we leave the validation of this to future work.

Programming Model. We want to determine how to configure a software router so as to combine programmability and performance; hence, we assume that our router is programmed using the Click programming model [8]—to our knowledge, the most successful research effort that sought to combine these two properties. More specifically, we assume that a packet-processing application is written as a set of modular "elements", each specifying a packet-processing task; hence, a router's functionality can be described by a data-flow graph, which specifies the sequence of elements traversed by each received packet. A simple example of a data-flow graph is shown in Figure 1(a).

Opportunities for Parallelism. When configuring a multicore software router, a key question is how to parallelize router functionality across the available processing cores, i.e., determine which part of the data-flow graph will be executed by each core. This "parallelization" has a dramatic impact on the router's performance—it is easy to imagine
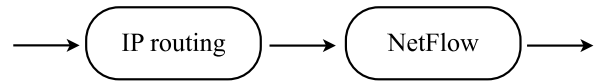
(a) A simple data-flow graph with two elements: IP routing and NetFlow. All packets received by the router are subjected first to IP routing, then NetFlow processing, before being forwarded to the next hop.
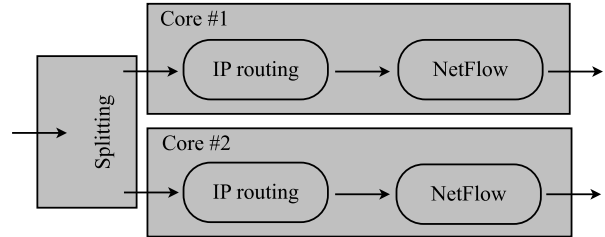
(b) The cloning approach: Incoming traffic is split among the two cores. Each core performs all processing for all packets it receives.
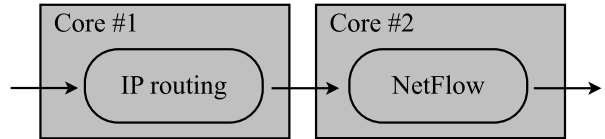
(c) The pipelining approach: Incoming packets traverse both cores. Each core performs a different kind of packet processing.

Figure 1: Two approaches to parallelizing router functionality across two processing cores.

how over- or underestimating the number of cores required for a particular task can lead to bad utilization of the router's resources.

There are two general approaches to performing this parallelization (illustrated in Figures 1(b) and 1(c)) [5].

- *Cloning*: In this approach, each core executes the entire data-flow graph. I.e., incoming traffic is split across all cores, and each core performs all the processing required for each packet it receives.

- *Pipelining*: In this approach, the data-flow graph is partitioned, and each part is assigned to a different core. I.e., each incoming packet traverses multiple cores, and each core performs a different processing task on the packet.

Recent work that uses multicore general-purpose hardware for packet processing follows the cloning approach, because it has been shown to yield higher performance for simple workloads, e.g., where the router performs IP routing on all packets [6, 7]. But is this always the right choice?

Consider, for instance, the example of Figure 1, where we want to run IP routing and NetFlow (a widely deployed router application that collects per-flow statistics [2]) on a server architecture with two cores. Suppose each core has an αMB last-level cache (LLC), while each application uses

a data structure (e.g., a forwarding table and a flow table, respectively) of approximately αMB. If we apply the cloning approach, each core runs both applications, which means that it accesses 2αMB of data through the αMB cache. In contrast, if we apply the pipelining approach, each core runs only one of the two applications, hence accesses its entire data structure from the cache. So, in this particular example, it is possible that the pipeline approach leads to fewer cache misses; and (depending on the server's bottleneck) fewer cache misses could translate to higher performance.

This simple, yet realistic scenario suggests that each approach—cloned or pipelined—may be the right choice (i.e., yield higher performance) in different scenarios. Extending this choice to multiple elements in a more complex data-flow, results in a large space of potential parallelizations—cloned, pipelined and various combinations of the two (i.e., where some portions of the data-flow are cloned and others are pipelined).

Our Approach to Parallelization. We want to answer the following question: given a multicore server architecture and a data-flow graph representing a set of packet-processing applications, which part of the graph should each core execute so as to maximize the router's performance?

We set out to design a decision process that takes as input (1) a server hardware profile; (2) a data-flow graph; and (3) a profile of each element in the data-flow graph; and outputs which element(s) should be run by each core. The server hardware profile should specify the set of available resources (e.g., number of processors, number of cores, cache sizes, etc.); the element profiles should specify the resource requirements for each element; the decision process should compare all the possible parallelization options (cloning, pipelining, or any combination of the two) and choose the option that maximizes the router's performance.

Several questions must be answered for this overall strategy to work. For example, what should a server hardware profile and an element profile look like, in the sense of what are the right measures to include in them, can/should these measures be compiled statically or at run-time, and so forth. We leave these (admittedly non-trivial) questions to future work. Our focus in this paper is on the decision process and in particular on understanding the trade-offs between the cloning and pipelining approaches which is at the core of any decision process.

Hence, as a first step, in this paper, we focus on the question: assuming perfect knowledge of the server architecture and the applications running on it (i.e., the elements of the data-flow graph), what factors must we consider in making the cloning vs. pipelining decision? This boils down to understanding when and how the two approaches differ in the overheads they incur (e.g., cache misses, synchronization overheads) under different scenarios; implicitly, understanding these overheads also reveals the extent of performance improvement that we stand to gain by making the "right" decision.

3. CONTROLLING PARALLELISM

3.1 Parallelization Overheads

We start by identifying the overheads associated with different parallelization options. Note that, since our goal is to make the cloning-vs-pipelining decision, we need only focus on *distinguishing* overheads—i.e., overheads incurred under cloning but not pipelining, and vice versa. We identify three such overheads: due to *synchronization*, *cache contention*, and *pipeline imbalance*; we describe each one and present a simple experiment that justifies our statements.

Experimental Setup. We use a general-purpose server, equipped with two quad-core Intel Nehalem Xeon 5560 processors; the four cores of each processor share an 8MB L3 cache, while each core has a private 256KB L2 cache and a private 32KB L1 cache. The server is also equipped with two network cards, each with two 10Gbps ports. In all presented experiments, the server is fed a workload of 64-byte packets, because this is the workload for which the server's performance is most affected by the cloning-vs-pipelining decision (we explain why below). Traffic arrives at all 4 ports at the same rate; all traffic received at one port is sent out the other port of the same card. The server runs SMP Click [5].

We use a data-flow graph with two processing elements, similar to the one of Figure 1(a) (the only difference is that instead of IP forwarding and NetFlow, we use synthetic packet-processing applications). Each received packet is processed by both elements. For each received packet, element i (where $i = \{1, 2\}$) reads the packet headers, then reads N_i random memory locations from an array of size S_i bytes, where N_i and S_i are configurable parameters. As we will see, although simple, this functionality captures those aspects of packet-processing applications that determine whether cloning or pipelining is the right choice.

When we use the cloning approach, each of the 8 cores receives an 8th of incoming traffic, executes the functionality of both elements, and sends the packets out in the network. When we use the pipelining approach, processor 1 receives the incoming traffic and executes the functionality of the first element, then passes the packets to processor 2, which executes the functionality of the second element and sends the packets out in the network; this is done such that each core handles a fourth of the traffic.

In all experiments we present, whether we use the cloning or pipelining approach, the server is bottlenecked at the CPU (i.e., the processing cores run out of cycles) and most cycles are spent waiting for memory accesses to complete—which makes sense, since our elements do nothing but read from memory.

Synchronization. With pipelining, each packet is processed by multiple cores that do not share a cache. This requires synchronization between multiple cores due to the following operations:

1) *Metadata sharing*: In Click, one core "passes" a packet to another core by storing a pointer to the packet's socket buffer descriptor (the Linux-kernel data structure that contains the packet's metadata, e.g., where the packet is stored in memory) in a first-come-first-serve queue served by the second core. These enqueuing and dequeuing operations require extra cycles. Moreover, there is an extra cache miss when the second core reads the pointer. Finally, book-keeping the queue (e.g., when the cores check whether the queue is full/empty) leads to extra cache misses when the status of the queue has changed.

2) *Content sharing*: Every time a core processes a new packet, it reads the contents of the corresponding socket buffer descriptor and the packet headers. Moreover, if the core performs deep-packet inspection, it also reads the contents of the packet. Each of these read operations results in cache misses when the socket buffer descriptor and the packet do not reside in a cache accessible by the core. With cloning, each packet is processed by one core, hence these read operations are performed once. In contrast, with pipelining, these read operations are performed every time a packet is passed from one core to another, resulting in more cache misses and lower router performance.

3) *Memory recycling*: In Click, each core that reads a packet from the network (the "receiving" core) stores it in a pre-allocated memory pool, and there is a separate pool for each core. With cloning, the receiving core is the one that also sends the packet out into the network (the "transmitting" core); hence, once the packet is transmitted, the core can easily recycle the corresponding packet buffer back into its pool. In contrast, with pipelining, the receiving core is different from the transmitting core; hence, the transmitting core has two options for recycling the packet buffer: into the receiving core's pool, which requires synchronization between the two cores for accessing the same pool; or, into the transmitting core's pool, which will cause the receiving core to exhaust its pool and re-allocate memory—a costly operation in terms of cycles.

We identified these as the main sources of synchronization overhead by running the following experiment. We configured the two processing elements with parameter values $S_1 = S_2 = 2$MB and $N_1 = N_2 = 1$ memory access per packet, i.e., for each received packet, each element reads one random memory location from a 2MB array. We picked S_i such that the data structures needed by both elements fit comfortably in our L3 cache, in order to avoid cache misses due to cache contention. Given this setup, the router achieves throughput 8.56Gbps with cloning and 2.89Gbps with pipelining, i.e., with pipelining, throughput drops by 66%. To understand this drop, we profiled our router under each approach. We found that cloning results in about 4 L3 cache misses per packet (two associated with the socket-buffer descriptors and two with the packet headers), whereas pipelining results in 13–14 additional cache misses per packet (which correspond to the read operations mentioned above).

We conclude that pipelining introduces significant synchronization overhead, hence we expect cloning to perform better in scenarios where compulsory cache misses are the dominant cost factor (in terms of cycles spent per packet).

Cache Contention. With cloning, each core runs all elements, which means that all elements contend for the cache (at all levels). This can lead to cache contention, when the aggregate working set size of the elements does not fit in the LLC (in our setup, the L3 cache). In contrast, pipelining allows different elements to use different L3 caches, which should reduce cache contention. To understand the impact of cache contention on router performance, we repeated the last experiment multiple times, for different S_i and N_i values, and measured the "benefit" of pipelining over cloning, i.e., the router's throughput under pipelining divided by the router's throughput under cloning. Hence, a benefit larger than 1 means that pipelining performs better than cloning, whereas a benefit below 1 means the opposite.
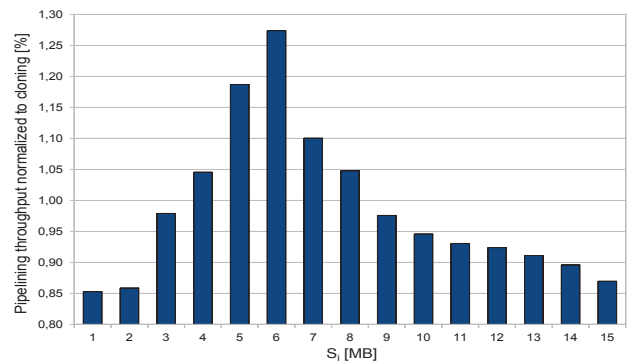
(a) $N_i = 200$.
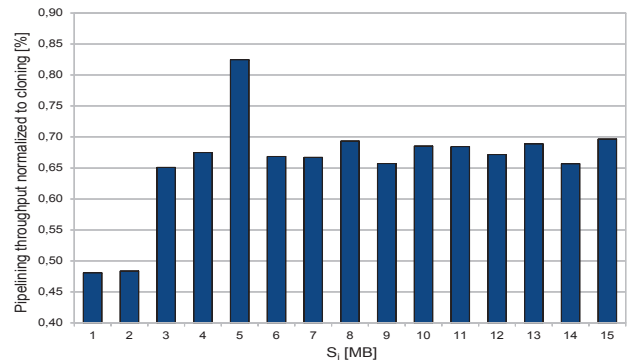
(b) $N_i = 20$.

Figure 2: Cache contention overhead.

Figure 2(a) shows the benefit of pipelining over cloning for $N_i = 200$ memory accesses per packet, as a function of S_i; we see that the benefit of pipelining increases with S_i until it reaches a maximum point (at $S_i = 6$MB), then it decreases. We explain this as follows. For small numbers of S_i, the arrays of both elements fit in the L3 cache, hence neither cloning nor pipelining incur cache contention; due to the synchronization overhead of pipelining, cloning performs

better. Beyond some point ($S_i = 3$MB), one element's array still fits in the L3 cache, whereas both arrays together do not, hence, cloning incurs cache contention whereas pipelining does not, and pipelining performs better. However, beyond a second point ($S_i = 6$MB), even one element's array does not fit in the L3 cache, hence, pipelining cannot avoid cache contention either, which causes its benefit to decrease. Finally, beyond a third point ($S_i = 9$MB), cache contention under pipelining becomes so strong that the synchronization overhead of pipelining outweighs its cache-contention benefit over cloning, and cloning starts performing better again.

Figure 2(b) shows the benefit of pipelining over cloning for $N_i = 20$ memory accesses per packet, as a function of S_i; we see that cloning always performs better (the benefit of pipelining over cloning is below 1) independently from S_i. This is because, when memory accesses are relatively infrequent, the L3 cache manages to absorb them to some extent, such that the cache-contention benefit of pipelining is outweighed by its synchronization overhead. We argued above that pipelining results in additional cache misses every time a packet is passed to a core that uses a different L3 cache (once, in our setup); hence, pipelining outperforms cloning, only if the latter introduces more additional misses due to cache contention.

We conclude that pipelining does reduce cache contention relatively to cloning, however, this benefit is easily outweighed by its synchronization overhead. Hence, we expect pipelining to perform better than cloning, only in scenarios that involve large, contention-sensitive data structures and frequent memory accesses.

Pipeline Imbalance. To work well, pipelining must have balanced pipeline stages, otherwise CPU cycles are wasted (since the packet rate through the pipeline is gated by the slowest stage). To demonstrate this effect, we configured the two elements with parameter values $S_1 = S_2 = 6$MB and $N_1 = 396$, $N_2 = 4$ memory accesses per packet. Given this setup, the router achieves throughput 728Mbps with cloning and 375Mbps with pipelining, i.e., cloning increases throughput by a factor of 2. This is because, with pipelining, 4 cores (those that run the first element) perform 99% of the work.

We conclude that, even when we have large, contention-sensitive data structures, we expect cloning to outperform pipelining if the pipeline stages are imbalanced.

In summary, we have identified three overheads that can lead to significant performance differences between a cloned vs. pipelined parallelization of the same data-flow. The nature of the identified overheads (they all consist of extra per-packet cycles) means that they affect performance only when the server is bottlenecked at the CPU—this is why we presented results obtained with a small-packet workload.

3.2 Problem Formulation

We now turn to the question of how we should combine the competing factors identified in the previous section to formulate the problem of choosing the "right" parallelization approach.

Consider a data-flow graph that consists of one sequence of n packet-processing elements (i.e., each packet must be sequentially processed by each element). Assume that elements do not share data structures other than socket buffer descriptors and packet headers and payloads. Element i is characterized by the number of cycles per packet consumed when running in isolation $C(i)$, the size of its data structures $S(i)$, and the number of memory requests issued per packet $N(i)$.

We want to implement this data flow on a server with m processors, each with multiple cores, a budget of B cycles, and a separate LLC of size LLC. We assign elements to processors, not individual cores, i.e., we only allow configurations where: if one core of processor j runs an instance of element i, then all cores of processor j run an instance of element i. Hence, when we say that "processor j runs element i," we mean that each core of processor j runs an instance of element i, and all cores share the same data structure accessed through the processor's LLC. Moreover, we impose the following restriction: any packet received by processor j is processed by all the elements on processor j. We define the "element placement" matrix A, of dimensions $n \times m$, as follows:

$$A_{ij} = \begin{cases} 1, & \text{if processor } j \text{ runs element } i \\ 0, & \text{otherwise.} \end{cases}$$

We denote by C_j the aggregate number of cycles per packet consumed by processor j, and by $S_j = \sum_i A_{ij} \cdot S(i)$ the aggregate data-structure size of the elements run by processor j.

Incoming traffic is load-balanced across all processors that run element 1. When processor j receives a packet to be processed according to element i, it does so, then: if $i = n$, sends the packet out; else, if processor j runs element $i + 1$, it processes the packet according to element $i + 1$; else, it passes the packet to a processor that runs element $i + 1$ (if there are many such processors, it load-balances the traffic across them). Load-balancing across processors happens based on processor throughput, i.e., processor j gets a fraction of the traffic that is inversely proportional to C_j.

We want to identify the element-placement matrix A that maximizes the throughput (in packets per second) of the router R:

$$R = \min_i \left\{ \sum_j \left\{ \frac{B}{C_j} \cdot A_{ij} \right\} \right\} \qquad (1)$$

This is essentially the throughput of the "slowest" element, i.e., the one that processes the fewest packets per second.

We make three simplifying assumptions: (1) Every LLC miss costs O_m cycles. (2) Inter-processor synchronization (i.e., passing a packet from one processor to another) costs O_s cycles. (3) When element i run by processor j accesses its data structure, the probability of a LLC miss is $\pi_j = 1 -$

$\frac{LLC}{S_j}$. I.e., we assume that each of the S_j memory locations served by the processor's LLC is equally likely to be in the cache.

Based on these three assumptions, the term C_j (the number of cycles per packet consumed by all elements run by processor j) can be broken into three components:

$$C_j = \sum_i C(i) \cdot A_{ij} + O_j^s + O_j^c$$

The first one is the aggregate number of cycles needed by all the elements run by processor j to process a packet, while the other two represent the synchronization and cache-contention overhead, respectively. We approximate the synchronization overhead as

$$O_j^s = \sum_{i=2}^n A_{ij} \cdot (1 - A_{i-1j}) \cdot O_s,$$

where $A_{ij} \cdot (1 - A_{i-1j})$ is the number of times processor j gets each packet from another processor and O_s is the number of cycles spent for inter-processor synchronization. We approximate the cache-contention overhead as

$$O_j^c = \sum_i A_{ij} \cdot \pi_j \cdot N(i) \cdot O_m,$$

where $\pi_j \cdot N(i) \cdot O_m$ is the expected number of cycles spent dealing with cache misses due to contention.

In summary, Eq. 1 specifies the throughput of the router, in scenarios where the bottleneck is the CPU (i.e., the server starts dropping packets because it does not have enough processing cycles to perform the necessary per-packet computation and/or memory accesses). Maximizing this throughput requires a closed formula for C_j; we build one, assuming that cycles are spent either performing per-packet computation or dealing with cache misses resulting from inter-processor synchronization and cache contention.

4. FUTURE WORK

We have taken a small first step toward automating the decision of how to parallelize a packet-processing flow inside a software router so as to maximize throughput. Much work remains to be done before we achieve our goal:

Quantifying the Performance Gain. Once we validate our model in more complex scenarios than the ones presented in Section 3.1, the next question is whether careful parallelization is necessary. We will know by comparing the throughput achieved with the element placement recommended by our model to the throughput achieved by (1) pure cloning, (2) pure pipelining, and (3) "obvious" hand-crafted solutions, when considering realistic packet-processing applications like IP forwarding, statistics collection, intrusion detection, encryption, etc.

Our results so far indicate that, at least for shared-memory architectures like Nehalem, pure cloning achieves the highest throughput, except from the cases where we have frequent memory accesses to large, contention-sensitive data

structures *and* nearly perfectly balanced pipeline stages—so, we need to assess how realistic these cases are. On the other hand, the ever-increasing size of routing tables, access-control lists, etc., suggests that minimizing cache contention for data structures (hence, introducing some amount of pipelining) will become increasingly important in the near future.

Implementation and Evaluation. If we confirm that careful parallelization is worthwhile, then we need to arrive at a practical implementation and a comprehensive evaluation, i.e., generalize our problem formulation, understand how to put together server and application profiles, build a practical compiler (which includes providing a solution to the problem formulated in Section 3.2), and test it on different server hardware.

5. CONCLUSION

A large part of the appeal of software routers has been that they will be easy to program. Counter-balancing this, is the need to achieve high performance which may require paying attention to lower-level, potentially hardware-specific details such as cache misses. Our high-level goal is to achieve high performance without losing programmability by developing a compiler that automates the decision process of how to parallelize a data-flow. As a first step towards this, we understand what parallelization options are possible—we defined these as pipelining vs. cloning. Then we identify three key factors that determine which parallelization approach is desirable: inter-socket synchronization costs, cache contention for data structures and the "imbalance" in a data-flow. Finally, we propose a strawman optimization framework that takes as input a profile of server resources and a data flow element's resource consumption and outputs an optimal mapping of elements to cores based on weighing the relative impact of the above factors.

6. REFERENCES

[1] Cisco Carrier Routing System. http://cisco.com/en/US/products/ps5763/index.html.
[2] Cisco IOS NetFlow. http://www.cisco.com/web/go/netflow.
[3] Vyatta Series 2500. http://vyatta.com/downloads/datasheets/vyatta_2500_datasheet.pdf.
[4] Vyatta Series 3500. http://vyatta.com/downloads/datasheets/vyatta_3500_datasheet.pdf.
[5] B. Chen and R. Morris. Flexible Control of Parallelism in a Multiprocesor PC Router. In *Proceedings of the USENIX Annual Technical Conference*, 2001.
[6] 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 *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2009.
[7] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-accelerated Software Router. In *Proceedings of the ACM SIGCOMM Conference*, 2010.
[8] E. Kohler, R. Morris, B. Chen, J. Jannoti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.
[9] Y. Ma, S. Banerjee, S. Lu, and C. Estan. Leveraging Parallelism for Multi-dimensional Packet Classification on Software Routers. In *Proceedings of the ACM SIGMETRICS Conference*, 2010.