# Toward Predictable Performance in Software Packet-Processing Platforms

*Mihai Dobrescu*
*EPFL, Switzerland*

*Katerina Argyraki*
*EPFL, Switzerland*

*Sylvia Ratnasamy*
*UC Berkeley*

## Abstract

To become a credible alternative to specialized hardware, general-purpose networking needs to offer not only flexibility, but also predictable performance. Recent projects have demonstrated that general-purpose multicore hardware is capable of high-performance packet processing, but under a crucial simplifying assumption of uniformity: all processing cores see the same type/amount of traffic and run identical code, while all packets incur the same type of conventional processing (e.g., IP forwarding). Instead, we present a general-purpose packet-processing system that combines ease of programmability with predictable performance, while running a diverse set of applications and serving multiple clients with different needs. Offering predictability in this context is considered a hard problem because software processes contend for shared hardware resources—caches, memory controllers, buses—in unpredictable ways. Still, we show that, in our system, (a) the way in which resource contention affects performance is predictable and (b) the overall performance depends little on how different processes are scheduled on different cores. To the best of our knowledge, our results constitute the first evidence that, when designing software network equipment, flexibility and predictability are not mutually exclusive goals.

## 1 Introduction

In recent years, both practitioners and researchers have argued for building evolvable networks, whose functionality changes with the needs of its users and is not tied to particular hardware vendors [4,6,17,22]. An inexpensive way of building such networks is to run a network-programming framework like Click [21] on top of commodity general-purpose hardware [6,17,22]. Sekar et al. recently showed that, in such a network, operators can reduce network provisioning costs by up to a factor of 2.5 by dynamically consolidating middlebox functionality, i.e., assigning packet-processing tasks to the available general-purpose devices so as to minimize resource consumption [26].

To become a credible alternative to specialized hardware, general-purpose networking needs to offer not only flexibility but also predictable performance: network operators are unlikely to accept the risk that an unlucky configuration could cause unpredictable drop in network performance, potentially leading to customer dissatisfaction and violations of service-level agreements. Several projects have demonstrated that general-purpose multicore hardware can perform packet processing at line rates of 10Gbps or more [16–18,22,23]. However, in all cases, this was achieved under a crucial simplifying assumption of uniformity: all processing cores see the same type/amount of traffic and run identical code, while all packets receive the same kind of conventional packet processing (e.g., IP forwarding or some particular form of encryption). This setup allowed for low-level tuning of the entire system to one particular, simple, uniform workload (e.g., manually setting buffer and batch sizes).

Building a general-purpose system that offers predictable performance is considered a hard problem, especially when this system needs to support an evolvable set of applications that are potentially developed by various third parties. Such a system may perform as expected under certain conditions, but then a change in workload or a software upgrade could cause unpredictable, potentially significant, performance degradation.

One important reason for this lack of predictability is the complicated way in which software processes running on the same hardware affect each other: false sharing [8], unnecessarily shared data structures [9], and contention for shared hardware resources (caches, memory controllers, buses) [30]. The last factor, in particular, has been the subject of extensive research for more than two decades: researchers have been working on predicting the effects of resource contention since the appearance of simultaneous multithreaded processors [5,10,12,28,29,31,32], yet, to the best of our knowledge, none of the proposed models have found their way into practice. And packet-processing workloads create ample opportunity for resource contention, as they move packets between network card, memory controller and last-level cache, all of which are shared among multiple cores in modern platforms.

We set out to design and build a packet-processing system that combines ease of programmability with predictable performance, while supporting a diverse set of applications and serving multiple clients, each of which

may require different types/combinations of packet processing. To offer ease of programmability, we rely on the Click network-programming framework [21]. We do not introduce any additional programming constraints, operating-system modifications, or low-level tuning for particular workloads.

We present a Click-based packet-processing system, built on top of a 12-core Intel Westmere platform, that supports a diverse set of realistic packet-processing applications (Section 2). Given this setup, we first investigate how resource contention affects packet processing: does it cause significant performance drop? how does the drop depend on specific properties of the involved applications? (Section 3) Then we look at how to predict these effects in a practical manner (Section 4). We also explore whether it makes sense to use "contention-aware scheduling" [34], a technique that reduces the effects of resource contention by not scheduling together processes that are likely to contend for hardware resources (Section 5).

Our main contribution is to show that it is feasible to build a software packet-processing system that achieves predictable performance in the face of resource contention. We also show that contention-aware scheduling may not be worth the effort in the context of packet processing. More specifically, using simple offline profiling of each application running alone, we are able to predict the contention-induced performance drop suffered by each of the applications sharing our system, with an error smaller than 3%. Moreover, in our system, the maximum overall performance improvement achieved by using contention-aware scheduling is 2%—and that only in one particular corner case. We provide intuition behind these results, and we quantitatively argue that they are not artifacts of the Intel architecture, rather they should hold on any modern multicore platform.

We consider our results to be good news for all the ongoing efforts in general-purpose networking: To the best of our knowledge, they constitute the first evidence that, when designing software network equipment, flexibility does not have to come at the cost of predictability.

## 2 System Setup

In this section, after introducing our hardware setup, we describe the packet-processing applications that we use to evaluate our work (§2.1) and the software configuration of our platform (§2.2).

As a basis for our system, we use a 12-core general-purpose server, illustrated in Figure 1, running SMP-Click [11] version 1.7, on Linux kernel 2.6.24.7. Our server is equipped with two Intel Xeon 5660 processors, each with $6 \times 2.8$GHz cores and an integrated memory controller. The 6 cores of each processor share a 12MB

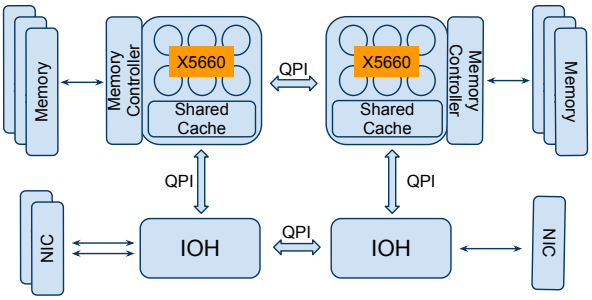



Figure 1: Overview of our platform's architecture.

L3 cache, while each core has private L2 (256KB) and L1 (32KB for instructions and 32KB for data) caches. The two processors are interconnected via a 6.4GT/sec QuickPath interconnect (QPI). The server has 6 DDR3 memory modules (2GB each, 1333MHz) and 3 dual-port 10Gbps network interface cards (NICs) that use the Intel 82599 Niantic [2] chipset, resulting in $6 \times 10$Gbps ports.

### 2.1 Workloads

We designed our workloads to involve realistic forms of packet processing but make the job of predicting their performance as hard as possible. We implemented 5 forms of packet processing that are deployed in current network devices and cover a wide range of memory and CPU behavior. In our experiments, we craft the traffic that is processed by the system so as to maximize resource contention. More specifically, we implemented the following types of packet processing:

▷ *IP forwarding (IP).* Each packet is subjected to full IP forwarding, including longest-prefix-match lookup, checksum computation, and time-to-live (TTL) update. We use the RadixTrie lookup algorithm provided with the Click distribution and a routing-table of 128 000 entries. As input, we generate packets with random destination addresses, because this maximizes IP's sensitivity to contention.

▷ *Monitoring (MON).* In addition to full IP forwarding, each packet is further subjected to NetFlow [1], a monitoring application. NetFlow collects statistics as follows: it applies a hash function to the IP and transport-layer header of each packet, uses the outcome to index a hash table with per-TCP/UDP-flow entries, and updates a few fields (a packet count and a timestamp) of the corresponding entry. As input, we generate packets with random IP addresses, such that the NetFlow hash table contains 100 000 entries. MON is a representative form of memory-intensive packet processing that benefits significantly from the L3 cache (both the routing table and the NetFlow hash table are cacheable data structures).

| Flow | cycles per instruction | L3 references per sec (millions) | L3 hits per sec (millions) | cycles per packet | L3 references per packet | L3 misses per packet | L2 hits per packet |
|---|---|---|---|---|---|---|---|
| IP | 1.33 | 25.85 | 20.21 | 1813 | 14.64 | 3.19 | 18.58 |
| MON | 1.43 | 27.26 | 21.32 | 2278 | 19.40 | 4.23 | 19.58 |
| FW | 1.63 | 2.71 | 2.13 | 23 907 | 20.22 | 4.29 | 56.10 |
| RE | 1.18 | 18.18 | 5.52 | 27 433 | 155.87 | 108.51 | 45.63 |
| VPN | 0.56 | 9.45 | 7.08 | 8679 | 25.63 | 6.41 | 30.71 |

Table 1: Characteristics of each type of packet processing during a solo run. Each number represents an average over 5 independent runs of the same experiment (the variance is negligible).

Also, it captures the nature of a wide range of packet-processing applications (applying a hash function to a portion of each packet and using the outcome to index and update a data structure).

▷ *Small firewall (FW).* In addition to full IP forwarding and NetFlow, each packet is further subjected to filtering: each packet is sequentially checked against 1000 rules and, if it matches any, it is discarded. We use sequential search (as opposed to a more sophisticated algorithm) because we consider a relatively small number of rules that can fit in the L2 cache. As input, we generate packets with random IP addresses that never match any of the rules; as a result, each packet is checked against all the rules, which maximizes FW's sensitivity to contention. This is a representative form of packet processing that benefits significantly from all the levels of the cache hierarchy.

▷ *Redundancy elimination (RE).* In addition to full IP forwarding and NetFlow, each packet is further subjected to RE [27], an application that eliminates redundant traffic. RE maintains a "packet store" (a cache of recently observed content) and a "fingerprint table" (that maps content fingerprints to packet-store entries). When a new packet is received, RE first updates the packet store, then uses the fingerprint table to check whether the packet includes a significant fraction of content cached in the packet store; if yes, instead of transmitting the packet as is, RE transmits an encoded version that eliminates this (recently observed) content. The assumption is that the device located at the other end of the link maintains a similar packet store and is able to recover the original contents of the packet. We implemented a packet store that can hold 1 second's worth of traffic and a fingerprint table with more than 4 million entries. This is a representative form of memory-intensive packet processing that does not significantly benefit from caching.

▷ *Virtual private network (VPN).* Each packet is subjected to full IP forwarding, NetFlow and AES-128 encryption. This is a representative form of CPU-intensive packet processing.

▷ *Synthetic processing (SYN).* For each received packet, we perform a configurable number of CPU operations (counter increments) and read a configurable number of random memory locations from a data structure that has the size of the L3 cache. We use this for profiling. We denote by *SYN_MAX* the most aggressive synthetic application that we were able to run on our system, which performs no other processing but consecutive memory accesses at the highest possible rate.

Table 1 summarizes the characteristics of each of these types of packet processing during a "solo" run (one core runs the packet-processing type, while all the other cores are idle). We use Oprofile [3] to count instructions, L2 hits, and L3 references and misses (we compute L3 hits as the difference between references and misses).

## 2.2 Software Configuration

**Packet-processing parallelization.** An important question is how packet processing should be parallelized among multiple cores. One possibility is the "pipeline" approach, where each packet is handled by multiple cores: one core reads it from memory, then passes it to another core for the first processing step, which passes it to another core for the second processing step, and so on. Another possibility is the "parallel" approach, where each packet is handled by a single core that reads the packet from memory and performs all the processing steps. The most recent general-purpose networking projects use the parallel approach, because it yields higher performance [16, 17]. However, a common criticism is that this is the case only for the simple, uniform workloads considered by these projects.

At first glance, choosing between the two approaches involves a trade-off (that we describe in detail in [14]). On the one hand, the parallel approach avoids passing the packet between different cores, hence eliminating synchronization and introducing fewer compulsory cache misses per packet. On the other hand, it requires that each core perform all the processing steps for each packet (hence accessing many different data structures), which may introduce a higher number of avoidable cache misses per packet due to cache contention. Hence, it seems intuitive that each approach would be best suited for different packet-processing applications.

After extensive experiments, we concluded that, in practice, there is no real trade-off between the two approaches: the parallel one is always better. This is because pipelining introduces several kinds of overhead that end up outweighing its potential benefit. For instance, passing socket-buffer descriptors, packet headers, and, potentially, payload between different cores results in compulsory cache misses. A less obvious source of overhead is memory management: Each core that handles packet reception uses a pre-allocated memory pool for storing packets. In a pipelined configuration, a packet is received by one core and transmitted by another; the transmitting core must recycle the buffer into the receiving core's pool of free buffers, and this requires extra synchronization between the two cores when removing/placing buffers in the pool. In our system, pipelining results in 10–15 extra cache misses per packet.

It *is* possible to craft a synthetic workload that performs better under the pipeline approach: it has to be a workload with enough processing steps and the right size of cacheable data structures such that running it on a parallel configuration results in more than 15 extra avoidable cache misses per packet than running it on a pipelined one. We describe such a workload in [14]: each received packet triggers more than 200 random memory accesses to a data structure that is *exactly* double the size of an L3 cache; even a small deviation from these numbers causes the advantage of the pipeline over the parallel approach to disappear. However, none of the realistic workloads that we looked at (including applications that involve deep packet inspection or redundancy elimination [27]) comes even close to such behavior.

*Our configuration.* We adopt the parallel approach for our system. Traffic arriving at each of our $N$ network ports is split into $Q$ receive queues. We refer to all traffic arriving at one receive queue as a *flow*; this is traffic that corresponds to one set of clients of our networking platform, all of which require the same type of packet processing. Each flow is handled by one core, which is responsible for reading the flow's packets from their receive queue, performing all the necessary processing, and writing them to the right transmit queue. Each core reads from its own receive queue(s) and writes to its own transmit queue(s), which are not shared with other cores. So, we have $N \cdot Q$ flows, each one assigned to one core, and each flow potentially involving a different type of packet processing.

In this paper, we focus on the scenario where each core processes one packet-processing flow: we use $N = 6$ ports and $Q = 2$ receive queues per port, so we have 12 different flows, each assigned to a separate core (we discuss this choice in Section 6). However, the Niantic cards support up to $Q = 128$ receive queues, so our prototype can, in principle, support hundreds of different flows.

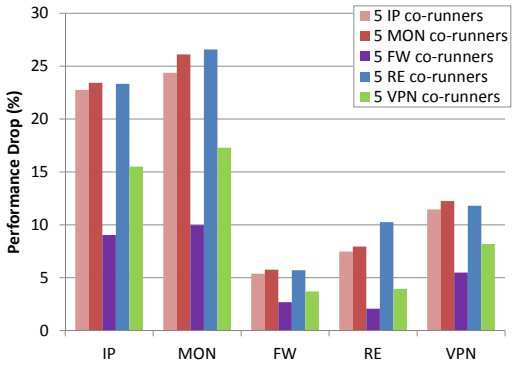
**NUMA memory allocation.** We ensure that each flow accesses its data "locally," i.e., through the memory controller that is directly connected to the processor handling the flow. We do this for two reasons: First, it has been shown (and we also verified experimentally) that accessing data remotely has a significant impact on memory-access latency [7], which, in our context, results in significant performance degradation. Second, to access data remotely, a flow has to use the processor interconnect, which can become a significant contention factor [7, 34].

Theoretically, there are two scenarios where we might not be able to ensure local memory access, and we explain next why these do not arise in our system:

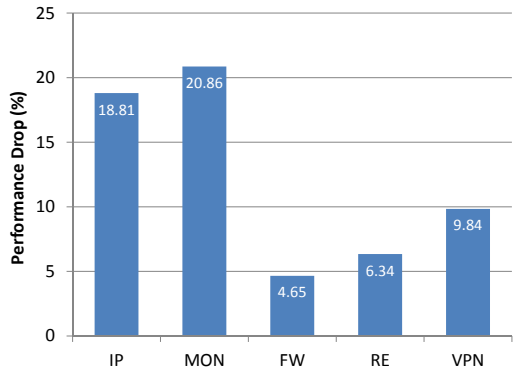
(a) Consider two flows, $f_1$ and $f_2$, that run on different processors and access the same data structure; one of the two flows will have to access the data remotely. This scenario does not arise in our system: If there are multiple flows that need to access the same data structure, we run all of them on the same processor. If there are more flows than per-processor cores that need to access the same data structure, we replicate the data structure across memory domains. We acknowledge that, in principle, such replication may break the semantics of a packet-processing application, however, we have not yet encountered any such (realistic) case. The closest we came was the redundancy-elimination application (described in §2.1), but, even there, it turned out that all the relevant data structures could be replicated across memory domains.

(b) Consider a flow $f$ running on a core of processor $P_1$, accessing its data locally; due to contention-aware scheduling [7, 34], we decide to move this flow to a core of processor $P_2$; as a result, $f$ must now access its data remotely. This scenario does not arise in our system either: we will argue that, in our context, it does not make sense to perform contention-aware scheduling—one of the resulting benefits is that we do not have to deal with remote memory accesses.

**Avoidable contention in the software stack.** Before setting out to study resource contention between applications running on different cores, we sought to eliminate any form of "underlying" resource contention from our system, i.e., contention introduced not by the applications themselves but by the design of the underlying software stack: NIC driver, operating system, Click. We identified (a) false sharing and (b) unnecessary data sharing among multiple cores (e.g., the book-keeping data structures in the Niantic driver and the random seed of the Click random number generator were shared among multiple cores) as sources of such contention. We eliminated the former by padding data structures appropriately and the latter by replicating per-core data structures. Similar problems and fixes were recently presented in a scalability analysis of the Linux kernel [9].




(a) Performance drop suffered by each flow type in each scenario. For example, when a MON flow co-runs with 5 RE competitors, it suffers a drop of 27%.




(b) Average performance drop suffered by each flow type across all 5 scenarios that involve a target flow of that type. For example, the average performance drop suffered by the MON flow across all 5 scenarios (that involve a target MON flow) is 20.86%.

Figure 2: The effect of resource contention. For each pair of realistic flow types $X$ and $Y$, we run an experiment in which a flow of type $X$ co-runs with 5 flows of type $Y$. We measure the performance drop suffered by the flow of type $X$.

# 3 Understanding Contention

In this section, we identify which resources packet-processing flows mostly contend for (Section 3.1) and which flow properties determine the level of contention (Section 3.2), and we provide intuition behind our observations (Section 3.3).

We observe that the contention-induced performance drop suffered by a packet-processing flow is mostly determined by the number of last-level cache references per second performed by other flows sharing the same cache—not so much by the particular types of packet processing performed by these flows. As we will see, this observation is what enables us to do simple yet accurate performance prediction (Section 4).

In terms of terminology and notation, when we use the term "cache," we refer to the last-level cache shared by all cores of the same processor unless otherwise specified. We use "cache refs/sec" as an abbreviation for "cache references per second." We say that a flow *co-runs* with other flows when they all run on different cores of the same processor; we refer to all these flows as *co-runners*. In each experiment, we typically co-run 6 flows and study the performance drop suffered by one of these flows due to contention; we denote this flow by $T$ (for "target") and each of its co-runners by $C$ (for "competitor"). With respect to a flow $T$, we use the term *competing references* to refer to all the last-level cache references performed by this flow's co-runners.

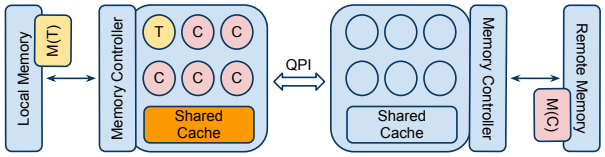
In all our experiments, we compute the performance drop suffered by a flow $T$ due to contention with a set of competing flows as follows: First, we measure the throughput $\tau_s$ achieved by flow $T$ during a solo run. Then we measure the throughput $\tau_c$ achieved by flow $T$ when it co-runs with the set of competing flows. The contention-induced performance drop suffered by flow $T$ is $\frac{\tau_s - \tau_c}{\tau_s}$. Each data point in our graphs represents the average over 5 independent runs of the same experiment (the variance is negligible).

We start by measuring the contention-induced performance drop suffered by realistic flow types in different scenarios (Figure 2). MON is the most sensitive type, suffering a performance drop of up to 27% (highest bar in Figure 2(a)), while FW suffers less than 6% in all experiments. RE is the most aggressive flow type, causing a performance drop of up to 27%, while FW causes a performance drop of less than 10% in all experiments. To draw meaningful conclusions from these numbers, we need to understand what are the properties of a packet-processing flow that make it sensitive and/or aggressive with respect to contention.

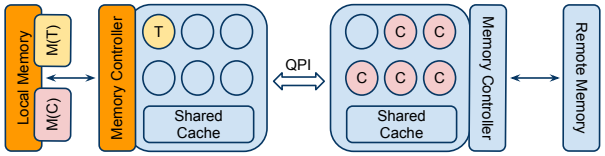
## 3.1 Contended Resources

We first identify which resources are responsible for contention. There are two candidates: the cache and the memory controller. We can rule out the processor interconnect, because our configuration ensures that each flow accesses its data locally, hence does not use this interconnect (Section 2.2).

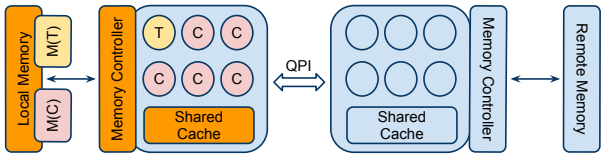
To assess the level of contention for the two candidate resources, we use the three system configurations illustrated in Figure 3. Each configuration allocates processing cores and memory to the co-running flows, so as to expose contention for different resources [7]: the first configuration creates contention only for the cache, the second one only for the memory controller, and the third one for both. In each configuration, we measure the contention-induced performance drop suffered by re-




(a) $T$ contends with $C$s only for the L3 cache. $C$s' data is remote, hence accessed through a different memory controller.




(b) $T$ contends with $C$s only for the memory controller. $C$s run on a different processor, hence use a different L3 cache.



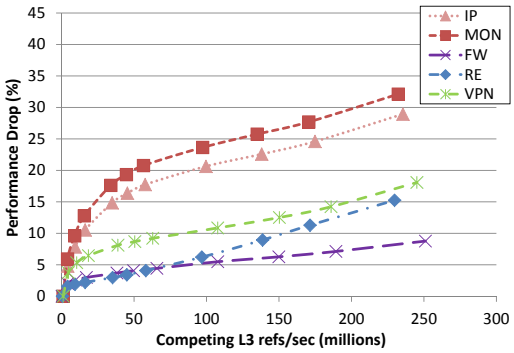

(c) $T$ contends with $C$s for both the memory controller and the L3 cache.

Figure 3: Configurations that expose contention for different resources. The resource that is contended in each configuration is highlighted. $T$ denotes the target flow (whose performance drop we are measuring) and $M(T)$ denotes flow $T$'s data structures. $C$ denotes a competing flow and $M(C)$ denotes the corresponding data structures.

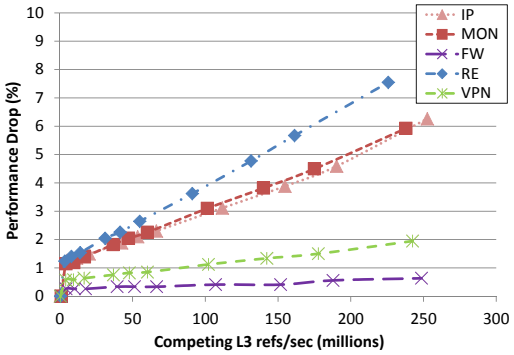
alistic flow types when they encounter different levels of competition. Figure 4 shows the drop suffered by each flow type when it co-runs with SYN flows, as a function of the cache refs/sec performed by the SYN flows.

These numbers show that the dominant contention factor is the cache. The most sensitive flow type (MON) suffers up to 32% when competing for the cache only (the curve with square data points in Figure 4(a)) and up to 6% when competing for the memory controller only (the curve with square data points in Figure 4(b)).

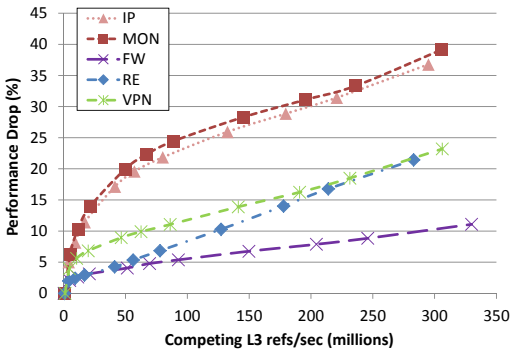
Our conclusion relates to prior work as follows: It differs from the conclusion drawn by running SPEC benchmarks on multicore platforms—in that case, the dominant contention factors were found to be the memory controller and the processor interconnect [7, 34]. The difference may come from the fact that packet-processing workloads benefit from the cache more than SPEC benchmarks and/or the fact that, in our context, overloading the interconnect is unnecessary. Our conclusion is consistent with recent results on software routers: in a software router running on an Intel Nehalem platform, as long as we have sufficient network I/O capacity, the bottleneck lies with the CPU and/or memory latency [16].




(a) Contention for the L3 cache. Performance drop suffered by each flow type in the configuration of Figure 3(a).




(b) Contention for the memory controller. Performance drop suffered by each flow type in the configuration of Figure 3(b).




(c) Contention for both resources. Performance drop suffered by each flow type in the configuration of Figure 3(c).

Figure 4: The effect of contention for different resources. For each realistic flow type $X$, we co-run a flow of type $X$ with 5 flows of type SYN multiple times, ramping up the number of cache refs/sec performed by the SYN flows. We measure the performance drop suffered by the flow of type $X$ as a function of the competing cache refs/sec.

## 3.2 Sensitivity and Aggressiveness

We now look at which properties of a packet-processing flow determine its sensitivity and aggressiveness, i.e., the amount of damage that it suffers from its co-runners and the amount of damage that it causes to them.

First, we observe a positive correlation between a flow's sensitivity to contention and the number of cache hits per second that it achieves during a solo run. Figure 2(b) shows that the higher the number of hits per second achieved by a flow type during a solo run (Table 1), the higher the average performance drop suffered by the flow. This makes sense: sharing a cache with co-runners causes memory references that would result in cache hits (if the flow ran alone) to become cache misses; the more hits per second a flow achieves during a solo run, the more opportunity there exists for these hits to become misses, leading to higher performance drop.

Second, we observe that the amount of damage suffered by a given flow is mostly determined by the number of competing cache refs/sec, not so much by the types of the competitors. Said differently, two flows, $C_1$ and $C_2$, that perform the same number of cache refs/sec will cause roughly the same performance drop to a given co-runner, regardless of whether $C_1$ and $C_2$ involve the same or different types of packet processing. This can be seen in Figure 5, which shows the performance drop suffered by different flows when they co-run with SYN as well as realistic competitors. For instance, a MON flow suffers a 27% drop when competing with 5 RE flows that generate 80 million cache refs/sec, and it suffers a 24% drop when competing with 5 SYN flows that generate the same number of cache refs/sec. So, RE flows cause about the same damage with SYN flows that generate the same rate of cache references, even though RE involves redundancy elimination, whereas SYN involves random memory accesses.

We found this observation partly intuitive and partly surprising: The intuitive part is that more competing cache references result in more damage, because they reduce the effective cache space of the target flow. The surprising part is that the particular memory access pattern of the competitors is not significant, and instead the rate of competing cache references mostly determines the amount of damage suffered by flows. It is worth noting that most of the complexity of existing mathematical models that predict contention effects comes from their effort to characterize the memory access patterns of the co-runners and their interaction.

Third, we observe that a sensitive flow's performance at first drops sharply with the number of competing cache refs/sec, however, beyond some point, the drop slows down significantly. For instance, as we see in Figure 5, a MON flow's performance drops by 20% when competition goes from 0 to 50 million cache refs/sec, but only an extra 5% when competition goes from 50 to 100 million cache refs/sec. As a result, a MON flow's performance drops roughly the same, whether it is co-running with IP, MON, or RE competitors, since all these flows contribute at least 50 million cache refs/sec.

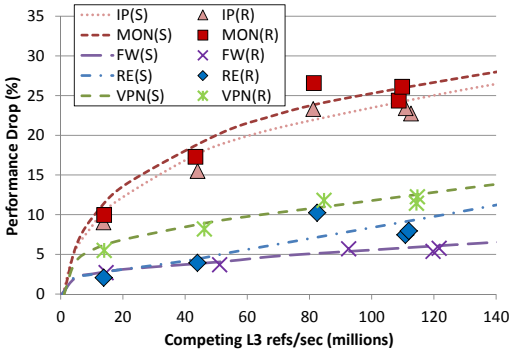



Figure 5: A merge of Figures 2(a) and 4(c). It shows the performance drop suffered by each flow type when it co-runs with SYN flows (curves) as well as realistic flows (individual points). For example, the curve MON(S) shows the performance drop suffered by a MON flow when it co-runs with SYN flows, while the individual squares MON(R) show the performance drop suffered by a MON flow when it co-runs with various realistic flow types. Each MON(R) square corresponds to a different realistic competitor type.

**Summary.** We made three observations: (a) A flow's sensitivity to resource contention depends on the number of hits/sec that the flow achieves during a solo run. (b) The specific amount of damage that a flow suffers due to contention is mostly determined by the number of cache refs/sec performed by its competitors, and not by the exact type of packet processing that they perform. (c) The performance of a sensitive flow at first drops sharply as the number of competing cache refs/sec increases; however, once a "turning point" is reached, the performance drop suffered by each sensitive flow stays within a relatively small range, no matter what type of co-runners it is competing with.

## 3.3 Explanation of our Observations

Before we use these observations, we provide intuition and potential explanations for them. Since the dominant contention factor is the cache, we concentrate on cache contention.

**Sensitivity depends on cache hits/sec.** We can express the performance drop suffered by a flow due to cache contention as follows:

- Suppose the flow achieves $h$ cache hits/sec and processes $n$ packets/sec during a solo run.
- Suppose that, due to contention, the flow suffers *hit-to-miss conversion rate* $\kappa$, i.e., each memory reference that was a hit during a solo run turns into a miss with probability $\kappa$.

- Without contention, processing $n$ packets takes 1 second. With contention, processing $n$ packets results in $\kappa \cdot h$ extra cache misses and takes $1 + \delta \cdot \kappa \cdot h$ seconds, where $\delta$ is the extra time needed to complete a memory reference that is a cache miss instead of a cache hit.

- Hence, the performance drop suffered by the flow in terms of packets/sec will be

$$\frac{n - \frac{n}{1+\delta\kappa h}}{n} = \frac{1}{1 + \frac{1}{\delta\kappa h}}. \quad (1)$$

Performance drop increases with competition (for the cache), primarily because the hit-to-miss conversion rate increases with competition. The value of $\delta$ provided by our platform's specs is 43.75 nanoseconds—although, in practice, its exact value depends on the nature of memory accesses and also slowly increases with competition.

In the worst case, the hit-to-miss conversion rate is $\kappa = 1$, i.e., all of the cache hits achieved by the target flow during a solo run turn into misses due to contention. Figure 6 shows this worst-case performance drop as a function of the number of cache hits/sec achieved by the flow during a solo run, for different values of $\delta$. E.g., assuming $\delta = 43.75$ nanoseconds, if a packet-processing flow achieves fewer than 20 million cache hits/sec during a solo run, even if all the hits turn into misses, the flow's performance cannot drop by more than 47%.

As a side note, according to Equation 1, a flow's worst-case performance drop depends only on the hits/sec achieved by the flow during a solo run, not by other characteristics of the flow (such as cycles spent on computation or total memory references per second); this is what makes hits/sec a good metric for a flow's worst-case sensitivity to contention.

**Aggressiveness is determined by cache refs/sec.** We observed that, in our setup, the aggressiveness of a set of flows is mostly determined by their cache refs/sec: a set of realistic flows and a set of SYN flows that perform the same number of cache refs/sec cause roughly the same damage to a given target flow $T$.

We explain this as follows: Each of our realistic packet-processing flows accesses at least a few megabytes of data. When several of these flows co-run, they access a total amount of data that is significantly larger than the cache size, which causes them to access the cache close to uniformly. As a result, from the point of view of a target flow $T$ that shares the cache with these flows, they behave similarly to a set of SYN flows (that access the cache uniformly by construction).

In Section 6, we briefly discuss the scenario where the working-set sizes of the competing flows are relatively small (such that the cache is not saturated) and explain why we do not address that scenario in this paper.

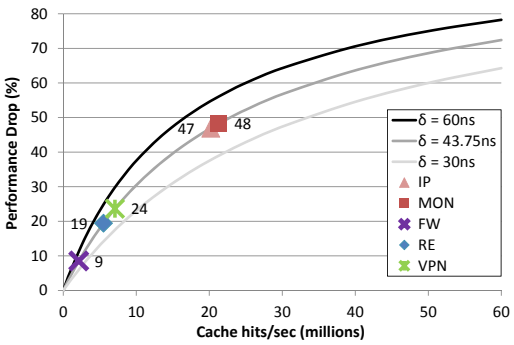



Figure 6: Estimated maximum performance drop suffered by a flow as a function of the cache hits/sec it achieves during a solo run. The estimates are based on Equation 1, for $\kappa = 1$ and different values of $\delta$. The graph also shows the data points that correspond to our realistic packet-processing flows, assuming $\delta = 43.75$ nanoseconds. For example, the maximum performance drop that could be suffered by an IP flow is 47%.

**Shape of the performance drop.** To understand how performance drop changes with competition, we look at how the hit-to-miss conversion rate changes with competition. Figure 7 shows the conversion rate suffered by a MON flow as a function of cache competition, as we measure it on our platform (using the configuration of Figure 3(a)) and as we analytically estimate it using a simple model. We will use the model to provide intuition (*not* accurate prediction), then discuss how it matches the measured data.

We describe the model in our technical report [15] and summarize here the gist: We have a target flow $T$ that shares a cache with a set of competitors.

- Consider a sequence of cache references, $\langle t, c_1, c_2, \ldots, c_Z, t' \rangle$, where: $t$ and $t'$ are two consecutive references performed by flow $T$ to the same cache line, $t'$ was a hit during a solo run, and $c_i, i = 1..Z$, are the competing references that occur between $t$ and $t'$.

- Suppose that each competing reference $c_i$ evicts the content cached by $t$ with probability $p_{ev}$, independently from any other competing reference. $t'$ is a hit if none of the $Z$ competing references evict this content, i.e., $P(hit|Z) = (1 - p_{ev})^Z$. The target flow's hit-to-miss conversion rate is $1 - P(hit)$.

- Suppose that each reference that occurs after $t$ is: either a competing reference, with probability $p_c$, or $t'$, with probability $p_t = 1 - p_c$. Hence, $Z$ is a random variable of geometric distribution with success probability $p_t$, i.e., $P(Z = z) = (1 - p_t)^z p_t$.

To compute $P(hit)$ as a function of competition, we need to know how $p_{ev}$ and $p_t$ change with competition. The following assumptions allow us to approximate them: (a) the competitors access the cache uniformly, (b) the target flow accesses its data uniformly, and (c) the target flow and the competitors have similar sensitivity to contention, i.e., suffer approximately the same hit-to-miss conversion rate.

Figure 7 shows that the shape of a flow's conversion rate as a function of competition can be explained as the result of basic cache sharing: The model-derived curve has a shape similar to the empirically derived curve (sharp rise at first, significant slow-down beyond some point), even though the model provides basic probabilistic analysis of cache sharing without considering any special feature of our platform. Note that, if we plug the model-derived conversion-rate values from Figure 7 into Equation 1 (for the value of $h$ that corresponds to a MON flow), we get an analytical estimate of a MON flow's performance drop as a function of competition, which also has a shape similar to the corresponding empirically derived curve.

Our model captures the shape, but overestimates the value of the conversion rate. This is because the model assumes that the target flow accesses its data uniformly, which is usually not the case. In Figure 7, we see that different MON functions are affected differently by contention: (a) "flow_statistics" suffers a conversion rate that is well captured by the model, which makes sense because the flow table is accessed uniformly. (b) "check_ip_header" and "skb_recycle" suffer insignificant conversion rates. Our explanation is that these functions reference the same few cacheable data with every received packet (e.g., book-keeping structures), so, their cacheable data is almost never evicted by their competitors. (c) "radix_ip_lookup" is somewhere in the middle. We think this is because the root of the radix trie and its children are "hot spots," i.e., they are accessed frequently enough to remain in the cache, whereas the rest of the trie does not have any such hot spots. However, for all functions, most of the hits that *are* susceptible to conversion are converted by the time competition reaches 50 million cache refs/sec—and our model does capture that effect.

As a side note, mathematical models that try to predict the effects of resource contention are complex because they try to analytically compute $p_{ev}$ and $p_t$ as a function of competition, and this is a hard task. Suppose the target flow performs $r_t$ cache refs/sec during a solo run. The competitors cause it to suffer extra misses that "slow it down," i.e., cause it to perform fewer than $r_t$ cache refs/sec; how much fewer depends on the competitors' cache refs/sec. At the same time, the target flow slows down the competitors by a degree that depends on the target flow's cache refs/sec. In the end, the relative

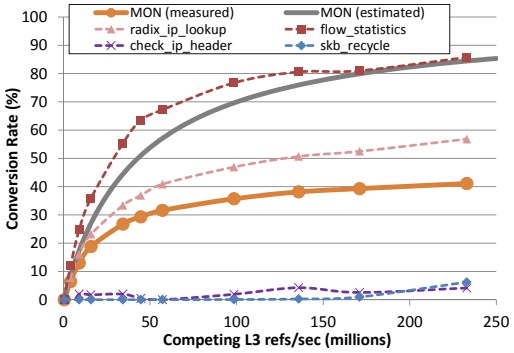



Figure 7: Measured and estimated hit-to-miss conversion rate suffered by a MON flow that shares the cache with SYN competitors, as a function of the competing cache refs/sec. The graph also shows the measured conversion rate suffered by separate functions of the MON flow. "flow_statistics" performs all the NetFlow-specific processing. "radix_ip_lookup" performs IP-table lookups. "check_ip_header" checks whether each packet has a valid IP header. "skb_recycle" performs memory management.

frequency of target and competitor memory references (which directly affects $p_t$) is the result of a complex interaction among the co-runners' particular access patterns. We were able to side-step this complexity (and crudely approximate $p_{ev}$ and $p_t$) because our goal was not to predict but merely to explain why increasing competition beyond some point does not significantly increase the resulting performance drop.

**Summary.** Our observations can be explained as the result of multiple processes sharing a last-level cache. This is not particular to our platform, but a universal artifact of modern server architectures.

## 4 Predicting Contention

In this section, we show how to accurately predict the overall and per-flow performance of our platform using simple profiling of each packet-processing flow running alone. Our prediction is based on the observation that a workload's aggressiveness is determined by the number of cache refs/sec that it performs, while it does not depend significantly on other workload properties.

Suppose we plan to co-run a flow $T$ with $|C|$ competing flows $C_1, C_2, ...C_{|C|}$. We predict flow $T$'s performance as follows:

1. We measure the number of last-level cache refs/sec $r_i$ performed by each flow $C_i$ during a solo run.

2. We co-run flow $T$ with different SYN flows, ramping up the number of cache refs/sec performed by

the SYN flows. We plot flow $T$'s performance drop as a function of the number of competing cache refs/sec.

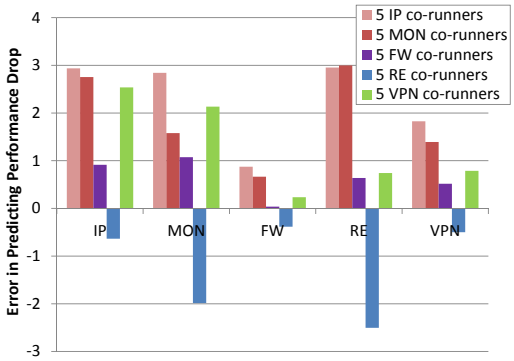
3. We predict that flow $T$'s performance drop will be equal to the value of the plot (derived at step #2) that corresponds to $\sum_{i=1}^{|C|} r_i$ competing cache refs/sec.

We rely on two assumptions. First, we assume that the competing flows will affect the flow $T$ as much as a set of SYN flows that perform the same number of cache refs/sec (this is well supported by the numbers in Figure 5). Second, we assume that each competing flow $C_i$ will perform as many cache refs/sec as it does during a solo run.

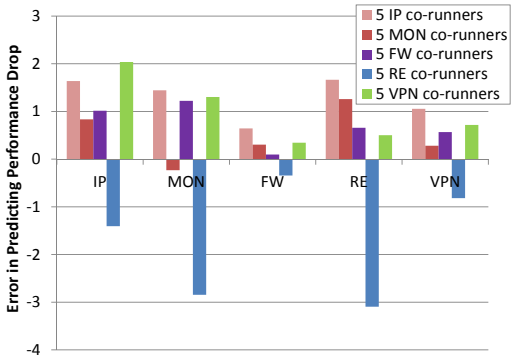
Our second assumption introduces a prediction error of a few percentage points: In reality, a competing flow $C_i$ that belongs to a sensitive type (e.g., IP or MON) will also suffer due to contention, hence its processing will slow down, resulting in fewer cache refs/sec than it performs during a solo run. By assuming that each competing flow $C_i$ will perform as many cache refs/sec as it does during a solo run, we overestimate the competition that flow $T$ will encounter, hence underestimate its performance. However, the resulting prediction error is small, because of the shape of the performance drop that we observed in Section 3: Once the number of competing cache refs/sec exceeds 50 millions or so, small changes in the number of competing cache refs/sec do not significantly change the damage to a sensitive flow. And sensitive flows like IP or MON (the ones whose number of cache refs/sec we overestimate) are also aggressive flows, i.e., they push the number of competing refs/sec beyond the 50 million turning point.

To validate our prediction method, we first reuse the workloads introduced in the beginning of Section 3: for each possible pair of realistic flow types $X$ and $Y$, we co-run a flow of type $X$ with 5 flows of type $Y$. We have already seen the performance drop suffered by each flow type in each of these scenarios (Figure 2); we will now look at how well we can predict these performance drops and how much of our error is due to each assumption. Figure 8(a) shows our prediction error, i.e., the difference between predicted and actual performance drop suffered by each flow type in each scenario. Figure 8(b) shows what the error *would* be, if we knew the exact number of competing cache refs/sec (we refer to this scenario as "prediction assuming perfect knowledge of the competition"). Figure 8(c) shows the absolute difference between predicted and actual performance drop suffered by each flow type, averaged across all scenarios.

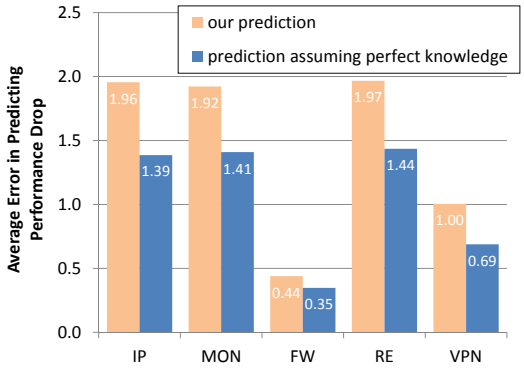
The average prediction error for each of the realistic flow types is less than 2% (tallest bars in Figure 8(c)). Our worst prediction errors are below 3%, and they correspond to the 2 leftmost bars in each group in Fig-




(a) Our prediction error. Difference between predicted and actual performance drop suffered by each flow type in each scenario.




(b) Prediction error assuming perfect knowledge of the competition. Difference between predicted and actual performance drop suffered by each flow type in each scenario, when we have perfect knowledge of the competing cache refs/sec.




(c) Average prediction error. Absolute difference between predicted and actual performance drop suffered by each flow type, averaged across all 5 scenarios that involve a target flow of that type. For example, we predict the performance drop suffered by a MON flow with an average error of 1.92% across all 5 scenarios (that involve a target MON flow).

Figure 8: Prediction errors for workloads with 2 flow types. For each pair of realistic flow types $X$ and $Y$, we run an experiment in which a flow of type $X$ co-runs with 5 flows of type $Y$. We measure/predict the performance drop suffered by the flow of type $X$.

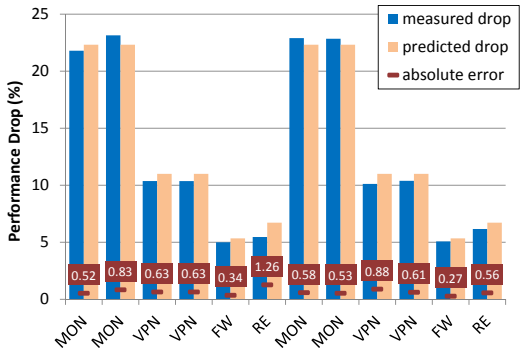



Figure 9: Prediction errors for a mixed workload. Predicted and actual performance drop suffered by each flow (and the absolute difference between the two).

ure 8(a): realistic flows that co-run with 5 IP or 5 MON competitors, respectively. In these scenarios, we overestimate the performance drop suffered by the target flow, partly because we assume that its co-runners will perform as many cache refs/sec as in the solo run. Actually, IP and MON are sensitive flow types that do suffer because of contention and perform fewer cache refs/sec compared to the solo run. The difference between the corresponding bars in Figure 8(a) and Figure 8(b) represents the error introduced by our second assumption. The rest of the error is due to our first assumption that the co-runners cause as much damage as a set of SYN flows that perform the same number of cache refs/sec.

We also validate our prediction method using a mixed workload: 2 MON, 2 VPN, 1 FW, and 1 RE flow per processor. Figure 9 shows the actual and predicted performance drop suffered by each flow, as well as the difference between the two. This time, we predict the performance drop suffered by each flow in the mix with a maximum error of 1.26%.

**Containing hidden aggressiveness.** Our prediction relies on offline profiling, i.e., running each packet-processing flow alone and measuring certain properties. However, it is possible that a flow (accidentally or on purpose) exhibits different behavior during offline profiling than during the actual run—a contrived example would be a flow that normally performs FW processing (i.e., is not aggressive), but, once it receives a specially crafted packet (potentially from an attacker), it switches mode and performs SYN_MAX processing (i.e., becomes very aggressive). Such a flow could mislead the system administrator into expecting significantly higher performance from her system and under-provisioning the system accordingly.

Nevertheless, a practical implication of our results is that an administrator can control the aggressiveness of each packet-processing flow simply by throttling the flow's rate of memory accesses. To verify this, we add to the beginning of each flow a "control element," which performs a configurable number of simple CPU operations, with the purpose of "slowing down" the flow and controlling the rate at which it performs memory accesses. At the same time, we monitor the rate at which each flow performs memory accesses using hardware performance counters and, if a flow exceeds the rate exhibited during its offline profiling, we configure its control element to slow it down accordingly.

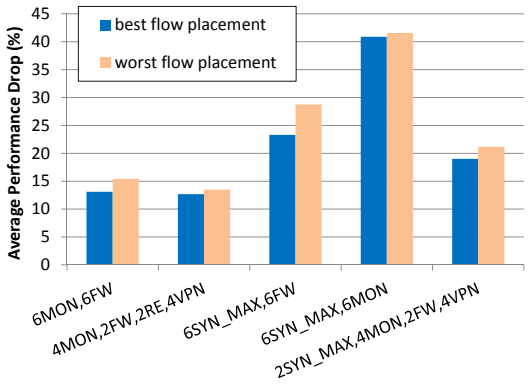
We tested this simple technique on our system and found that it ensures that each packet-processing flow performs no more than the profiled number of cache refs/sec. Thus, it is practical for an administrator to contain undue aggressiveness and achieve predictable performance.

## 5 Minimizing Contention via Scheduling

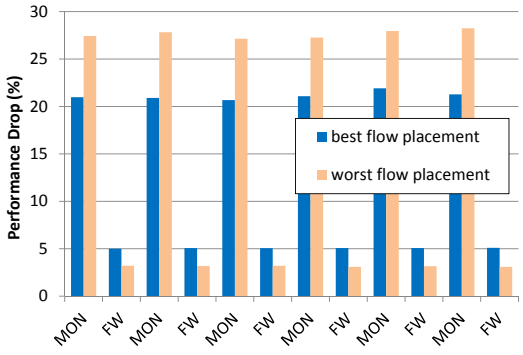
In this section, we explore the potential benefit of contention-aware scheduling [34] for packet-processing platforms. This is a family of techniques that solve the following problem: given $J$ processing jobs and a multicore platform with $J$ cores, how should we assign jobs to cores to minimize resource contention between the jobs and maximize the platform's overall performance? The basic idea at the core of the proposed solutions is to profile (offline or real-time) each process and avoid corunning aggressive with sensitive processing jobs.

To quantify the potential benefit of contention-aware scheduling for our system, we consider different combinations of 12 packet-processing flows. For each combination, we measure the contention-induced performance drop (averaged across all flows) under the worst and best flow-to-core placement (Figure 10(a)). The difference between these two numbers expresses the maximum we can gain in overall system performance through contention-aware scheduling.

For realistic-flow combinations, the maximum we can gain in overall system performance is 2% (Figure 10(a)). The flow combination for which we gain this maximum benefit is 6 MON and 6 FW flows. Figure 10(b) shows the per-flow performance drop for this combination, under the worst and best placement. The worst placement assigns the 6 MON flows to one processor and the 6 FW flows to the other, such that all the 6 MON flows (which are both aggressive and sensitive) have to compete with each other for the L3 cache; this causes a performance drop of 27% to each MON flow and an overall system performance drop of 15%. The best placement is the one that assigns 3 MON and 3 FW flows to each processor, such that each MON flow has to compete with only 2 other MON flows for the L3 cache; this causes a performance drop of 21% to each MON flow and an overall




(a) Average per-flow performance drop suffered under the worst and best placement, for different flow combinations.




(b) Per-flow performance drop suffered under the worst and best placement, for the 6-MON/6-FW combination.

Figure 10: Benefit of contention-aware scheduling. Performance drop suffered under the worst and best flow-to-core placement.

system performance drop of 13%. Hence, the extra damage introduced by the worst versus the best placement is 6% for each MON flow and 2% for the overall system.

Of all the possible realistic-flow combinations (given the flows that we implemented), this particular combination (6 MON and 6 FW flows) allows for the biggest overall improvement, because it is an equal mix of the most and least sensitive/aggressive flow types. One may think, at first, that a combination of more aggressive and/or sensitive flows (e.g., replacing the FW flows with IP or RE flows) would allow for a bigger improvement, but that is not the case: To create as big a difference as possible between the worst and best placement, we need a mix of sensitive, aggressive, and non-aggressive flows, such that in the worst placement sensitive flows co-run with the aggressive ones, whereas in the best placement sensitive flows co-run with the non-aggressive ones. Indeed, any other realistic-flow combination that we tried yielded an even smaller difference between worst and best placement.

This lack of (significant) difference between the worst and best placement makes sense, if we consider the observations in Section 3.2: once the competing cache refs/sec reach 50 millions or so, the performance drop suffered by a sensitive flow stays within a relatively small range, no matter which particular flows it co-runs with. Consider the 6-MON/6-FW combination: under the worst placement, each MON flow competes with 5 other MON flows, which generate about 100 million competing refs/sec, which causes the MON flow to suffer a performance drop of 27%; under the best placement, each MON flow competes with 2 other MON flows plus 3 FW flows, which generate about 60 million refs/second, which causes the MON flow to suffer a performance drop of 21%. In the end, as long as a placement generates more than a few tens of millions of cache refs/sec, it causes more or less the same performance drop to each sensitive flow.

If we consider non-realistic flows, the maximum we can gain in overall performance is 6%, for the 6 SYN_MAX, 6 FW combination (Figure 10(a)). SYN_MAX is the most aggressive and at the same time the most sensitive flow that we were able to craft (recall that it performs no processing other than memory accesses at the highest rate possible). So, even in the scenario where we have an equal mix of flows manifesting the most aggressive/sensitive behavior that we were able to generate in our system (SYN_MAX) and non-aggressive/non-sensitive flows (FW), the maximum benefit of contention-aware scheduling with respect to overall performance is 6%. Any other combination that we tried yielded an even smaller benefit.

## 6 Discussion

All the scenarios we considered have two common characteristics: each core runs a single packet-processing flow (Section 2.2) and the aggregate working-set size of the competing flows far exceeds the size of the cache (Section 3.3). If each core runs multiple flows, these compete for the L1 and L2 caches, so considering only the L3 accesses may not be sufficient to predict performance drop. If the working-set sizes of the flows are close to their fair share of the cache, then considering only the competing cache refs/sec may not be sufficient to characterize a workload's aggressiveness. These conditions may occur, for instance, in an active-networking setting, where large numbers of end users instantiate many small packet-processing flows on intermediate network elements.

We focused on one-flow-per-core, saturated-cache scenarios because we think that these are most likely to occur in the near future: State-of-the-art general-purpose platforms already offer tens of cores, and we consider

it unlikely that a network operator would need to support more than a few tens of different packet-processing types. Moreover, the point of building programmable packet-processing platforms is to make it easy to deploy new, interesting types of packet processing. All the emerging types of packet processing that we are aware of (e.g., redundancy elimination, deep packet inspection, application acceleration) would require several megabytes of frequently accessed data in a realistic network setting (e.g., a network interface that handles a few gigabits per second, located on the border of an Internet Service Provider). In state-of-the-art platforms, the size of the shared last-level cache is less than 3MB per core (and this will not increase in the near future, if the current architecture trends persist). Hence, we expect that running any combination of interesting packet-processing applications on a state-of-the-art multicore platform would saturate the shared caches.

## 7 Related Work

In recent years, we have seen a renewed interest in general-purpose networking, both by the industry [4] and the research community. Several research prototypes have demonstrated that general-purpose hardware is capable of high-performance packet processing (line rates of 10 Gbps or more), assuming simple, uniform workloads, where all the packets are subjected to one particular type of packet processing: IP forwarding [16], GPU-aided IP forwarding [17], multi-dimensional packet classification [23], or cryptographic operations [18]. Like all this work, our ultimate goal is to build high-performance software packet-processing systems. However, our focus here is to show that such a system can achieve *predictable* performance while running a wide range of packet-processing applications and serving multiple clients with different needs.

Researchers have been working for more than two decades on mathematical models for predicting the effects of resource contention. In the eighties and nineties, this was pursued in the context of general-purpose systems with simultaneous multithreading [5, 28, 31]. In the last decade, the focus has shifted to general-purpose multicore systems with shared caches [10, 12, 29, 32]. Zhang et al. recently questioned the need for prediction, with the argument that cache contention does not significantly affect the performance of modern parallel applications (in particular, PARSEC benchmarks) [33]. We show that, in the context of packet processing, resource contention can cause significant performance drop (up to 27%), however, we can accurately predict that without mathematical modeling. We should note that modeling does not remove the need for application profiling: all proposed models require as input at least the stack distance profile [24] of each application, which requires either instruction-set simulation of the application, or binary instrumentation and program analysis of the application, or co-running the application with a set of synthetic benchmarks [32].

A complementary topic to contention prediction is contention-aware scheduling: how to assign processes to cores so as to maximize overall system performance [7, 13, 19, 20, 25, 34]. We show that, in the context of packet processing, contention-aware scheduling does not significantly improve overall performance.

Finally, our work falls under the broader effort of exploring how software systems should be architected to exploit multicore architectures. That work has typically focused on redesigning software to expose parallelism—most recently by eliminating serial execution bottlenecks [9]. In contrast, we focus on packet-processing workloads, which are already amenable to parallel execution. Given a seemingly perfectly parallel system like a software packet-processing platform, we analyze what are the challenges involved in running such a system and—as a first step—what we can do to make its performance predictable.

## 8 Conclusion

We presented a software packet-processing system that combines ease of programmability with predictable performance, while supporting a diverse set of packet-processing flows. We showed that, in our system, we can accurately predict the contention-induced performance drop suffered by each flow (with an error smaller than 3%) thanks to two key observations: First, the performance drop suffered by a given flow is mostly determined by the number of cache references per second performed by its competitors, and not by the exact type of packet processing that they perform. Second, as long as the number of competing cache references per second exceeds a certain threshold, the performance drop suffered by a sensitive flow stays within a relatively small range, no matter what type of co-runners it is competing with. We also showed that, in our system, overall performance depends little on how different flows are scheduled on different cores, hence, contention-aware scheduling may not be worth the effort. We quantitatively argued that our results are not artifacts of a particular hardware architecture, rather they should hold on any modern multicore platform.

**Acknowledgments.** We would like to thank Aditya Akella, Ashok Anand, Michele Catasta, Jiaqing Du, Nikola Knezevic, Ovidiu Mara, Alexandra Olteanu, Simon Schubert, Vyas Sekar, our shepherd Srinivasan Seshan, and the anonymous reviewers for their help and constructive feedback.

## References


[1] Cisco IOS NetFlow. http://www.cisco.com/web/go/netflow.

[2] Intel 82599 10 GbE Controller Datasheet. http://download.intel.com/design/network/datashts/82599_datasheet.pdf.

[3] OProfile. http://oprofile.sourceforge.net.

[4] Why Use Vyatta? http://www.vyatta.org/getting-started/why-use.

[5] A. Agarwal, M. Horowitz, and J. Hennesy. An Analytical Cache Model. *Transactions on Computer Systems (TOCS)*, 7:184–215, 1989.

[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 *Proceedings of the ACM SIGCOMM Workshop on Programmable Routers for Extensible Services of TOmorrow (PRESTO)*, 2008.

[7] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A Case for NUMA-Aware Contention Management on Multicore Processors. In *Proceedings of the USENIX Annual Technical Conference*, 2011.

[8] W. J. Bolosky and M. L. Scott. False Sharing and its Effect on Shared Memory Performance. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, 1993.

[9] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[10] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2005.

[11] B. Chen and R. Morris. Flexible Control of Parallelism in a Multiprocesor PC Router. In *Proceedings of the USENIX Annual Technical Conference*, 2001.

[12] X. E. Chen and T. M. Aamodt. A First-Order Fine-Grained Multithreaded Throughput Model. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2009.

[13] G. Dhiman, G. Marchetti, and T. Rosing. vGreen: a System for Energy-Efficient Computing in Virtualized Environments. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2009.

[14] M. Dobrescu, K. Argyraki, M. Manesh, G. Iannaccone, and S. Ratnasamy. Controlling Parallelism in Multi-core Software Routers. In *Proceedings of the ACM SIGCOMM Workshop on Programmable Routers for Extensible Services of TOmorrow (PRESTO)*, 2010.

[15] M. Dobrescu, K. Argyraki, and S. Ratnasamy. Toward Predictable Performance in Software Packet-Processing Platforms. Technical report, Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland, 2012.

[16] 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.

[17] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-accelerated Software Router. In *Proceedings of the ACM SIGCOMM Conference*, 2010.

[18] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[19] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and Approximation of Optimal Co-Scheduling on Chip Multiprocessors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[20] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS Observations to Improve Performance in Multicore Systems. *IEEE Micro*, 28:54–66, 2008.

[21] E. Kohler, R. Morris, B. Chen, J. Jannoti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[22] G. Lu, C. Guo, Y. Li, Z. Zhou, T. Yuan, H. Wu, Y. Xiong, R. Gao, and Y. Zhang. ServerSwitch: A Programmable and High Performance Platform for Data Center Networks. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[23] 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.

[24] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9:78–17, 1970.

[25] A. Merkel, J. Stoess, and F. Bellosa. Resource-Conscious Scheduling for Energy Efficiency on Multicore Processors. In *Proceedings of the EuroSys Conference*, 2010.

[26] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[27] N. T. Spring and D. Wetherall. A Protocol Independent Technique for Eliminating Redundant Network Traffic. In *Proceedings of the ACM SIGCOMM Conference*, 2000.

[28] G. E. Suh, S. Devadas, and L. Rudolph. Analytical Cache Models with Applications to Cache Partitioning. In *Proceedings of the International Conference on Supercomputing (ICS)*, 2005.

[29] D. Tam, R. Azimi, L. B. Soares, and M. Stumm. Rapidmrc: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[30] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The Impact of Memory Subsystem Resource Sharing on Datacenter Applications. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2011.

[31] D. Thiebaud and H. S. Stone. Footprints in the Cache. *Transactions on Computer Systems (TOCS)*, 5:305–329, 1987.

[32] C. Xu, X. Chen, R. P. Dick, and Z. M. Mao. Cache Contention and Application Performance Prediction for Multi-Core Systems. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2010.

[33] E. Z. Zhang, Y. Jiang, and X. Shen. Does Cache Sharing on Modern CMP Matter to the Performance of Comtemporary Multithreaded Programs? In *Proceedings of the ACM Symposium on the Principles and Practice of Parallel Programming (PPoPP)*, 2010.

[34] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.