Building a power-proportional software router

Luca Niccolini[‡], *Gianluca Iannaccone*[†], *Sylvia Ratnasamy*[⋆], *Jaideep Chandrashekar*[§], *Luigi Rizzo*[‡][*]
[‡] *University of Pisa,* [†] *RedBow Labs,* [§] *Technicolor Labs,* [⋆] *University of California, Berkeley*

Abstract

We aim at improving the power efficiency of network routers without compromising their performance. Using server-based software routers as our prototyping vehicle, we investigate the design of a router that consumes power in proportion to the rate of incoming traffic. We start with an empirical study of power consumption in current software routers, decomposing the total power consumption into its component causes. Informed by this analysis, we develop software mechanisms that exploit the underlying hardware's power management features for more energy-efficient packet processing. We incorporate these mechanisms into Click and demonstrate a router that matches the peak performance of the original (unmodified) router while consuming up to half the power at low loads, with negligible impact on the packet forwarding latency.

1 Introduction

The network infrastructure is often viewed as an attractive target for energy-efficient design since routers are provisioned for peak loads but operate at low average utilization levels – *e.g.*, studies report network utilization levels of <5% in enterprises [28], 30-40% in large ISPs [15], and a 5x variability in ADSL networks [25]. At the same time, network equipment is notoriously inefficient at low load – *e.g.*, a survey of network devices [20, 24] reveals that the power consumed when a router is not forwarding *any* packets is between 80-90% of its peak power consumed when processing packets at full line rate. Thus although in theory networks offer significant opportunity for energy efficiencies, these savings are rarely realized in practice.

This inefficiency is increasingly problematic as traffic volumes continue their rapid growth and has led to growing attention in both research and industry [2, 9, 15, 16, 20, 24, 25, 28]. To date however, there have been no published reports on attempts to actually build an energy-efficient router. Motivated by this deficiency, and by recent advances in software routers [8, 12, 14, 17, 23], we thus tackle the question of how one might build an energy-efficient router based on general-purpose server hardware.

[*]This work was performed when the authors were affiliated with Intel Labs Berkeley.

The traditional challenge in building an energy efficient system lies in the inherent tradeoff between energy consumption and various performance metrics—in our case, forwarding rate and latency. We cannot compromise on peak forwarding rates since the incoming traffic rate is dictated by factors external to a given router (and since we do not want to modify routing protocols). We can however explore options that tradeoff latency for improved efficiency. The ability to do so requires: (1) that the underlying hardware expose primitives for low-power operation and (2) higher-level algorithms that invoke these primitives to best effect. Our work focuses on developing these higher-layer algorithms.

General-purpose hardware typically offers system designers three 'knobs' for low-power operation: (i) regulate the frequency at which individual CPU cores process work, (ii) put an idle core into a 'sleep' state (iii) consolidate packet processing onto fewer cores (adjusting the number of active cores).

As we shall see, not only do each of the above offer very different performance-vs-power tradeoffs, they also lend themselves to very different strategies in terms of the power management algorithms we must develop.

The question of how to best combine the above hardware options is, to our knowledge, still an area of active research for most application contexts and is entirely uncharted territory for networking applications. We thus start by studying the energy savings enabled by different hardware options, for different traffic workloads. Building on this understanding, we develop a unified power management algorithm that invokes different options as appropriate, dynamically adapting to load for optimal savings. We implement this algorithm in a Click software router [21] and demonstrate that its power consumption scales in proportion to the input traffic rate while introducing little latency overhead. For real-world traffic traces, our prototype records a 50% reduction in power consumption, with no additional packet drops, and only a small (less than 10 μs) cost in packet forwarding latency. To our knowledge, this is the first demonstration of an energy-efficient software router.

Before proceeding, we elaborate on our choice of general-purpose hardware as prototyping platform and how this impacts the applicability of our work. To some extent, our choice is borne of necessity: to our knowledge, current network hardware does not offer low-power modes of operation (or, at least, none exposed to third-party developers); in contrast, server hardware does incorporate such support, with standard software interfaces and support in all major operating systems. Therefore, our work applies directly to software routers built on commodity x86 hardware, an area of growing interest in recent research [12, 14, 17, 23] with commercial adoption in the lower-end router market [8]. In addition network appliances – load-balancers, WAN optimizers, firewalls, IDS, *etc.* – commonly rely on x86 hardware [1, 6] and these form an increasingly important component of the network infrastructure – *e.g.*, a recent paper [31] revealed that an enterprise network of $\sim$ 900 routers deployed over 600 appliances! However, at the other end of the market, high speed core routers usually employ very different hardware options with extensive use of network processors (NPs) or specialized ASICs rather than general purpose CPUs. Hence, our results cannot be directly translated to this class of network equipment, though we expect that the methodology will be of relevance.

The remainder of the paper is organized as follows. We start with an overview of related work (§2) and a review of the power consuption of current software routers (§3) and of the power management features modern server hardware offer (§4). We continue with the study of the tradeoffs between different power management options (§5). We present the design and implementation of our unified power management algorithm in §6 and the evaluation of our prototype in §7.

2 Related Work

We discuss relevant work in the context of both networking and computer systems.

Energy efficiency in networks. Prior work on improving network energy efficiency has followed one of two broad trajectories. The first takes a network-wide view of the problem, proposing to modify routing protocols for energy savings. The general theme is to re-route packets so as to consolidate traffic onto fewer paths. If such re-routing can offload traffic from a router entirely, then that router may be powered down entirely [18, 20].

The second line of work explores a different strategy: to avoid changing routing protocols (an area fraught with concerns over stability and robustness), these proposals instead advocate changes to the *internals* of a router or switch [11,16,28,30]. The authors assume hardware support for low-power modes in routers and develop algorithms that invoke these low-power modes. They evaluate their algorithms using simulation and abstract models of router power consumption; to date, however, there has been no empirical validation of these solutions.

Our focus on building a power-proportional software router complements the above efforts. For the first line of work, a power-proportional router would enable power savings even when traffic cannot be *entirely* offloaded from a router. To the second, we offer a platform for empirical validation and our empirical results in §5 highlight the pitfalls of theoretical models.

Energy-efficient systems. With the rise of data centers and their emphasis on energy-efficiency, support for power management in servers has matured over the last decade. Today's servers offer a variety of hardware options for power-management along with the corresponding software hooks and APIs. This has led to recent *systems* work exploring ways to achieve *power proportionality* while processing a particular workload.

Many of these efforts focus on cluster-level solutions that dynamically consolidate work on a small number of servers and power-down the remaining [13, 33]. Such techniques are not applicable to our context.

Closer to our focus is recent work looking at single-server energy proportionality [19, 22, 26, 27, 34]. Some focus on improved hardware capabilities for power management [19, 22]. The remaining [26, 27, 34] focus (like us) on leveraging existing hardware support, but do so in the context of very different application workloads.

Tsirogiannis *et al.* focus on database workloads and evaluate the energy efficiency of current query optimizers [34]. The authors in [27] focus on non-interactive jobs in data centers and advocate the use of system-wide sleep during idle times, assuming idleness periods in the order of tens of milliseconds. As such, their results are not applicable to typical network equipment that, even if lightly utilized, is never completely idle. More recently, Meisner *et al.* [26] explored power management trade-offs for online data-intensive services such as web search, online ads, *etc*. This class of workloads (like ours) face stringent latency requirements and large, quick variations in load. The authors in [26] use benchmarks to derive analytical models, based on which they offer recommendations for future energy-efficient architectures.

In contrast to the above, we focus on networking workloads and exploiting low-power options in current hardware. The result of our exploration is a lightweight, online, power saving algorithm that we validate in a real system. We leave it to future work to generalize our findings to application workloads beyond networking.

3 Deconstructing Power Usage

A necessary step before attempting the design of a power-proportional router is to understand the contribution of each server component to the overall power usage.

Server architecture. For our study, we chose an off-the-shelf server based on the Intel Xeon processor that is commonly used in datacenter and enterprise environments. Our server has two CPU processors each of which consists of six cores packaged onto a single die.

The two processors are Xeon 5680 "Westmere" with a maximum clock frequency of 3.3 GHz. They are connected to each other and to the I/O hub via dedicated point-to-point links (called QuickPath Interconnect in our case). The memory (6 chips of 1 GB) is directly connected to each of the processors. We equipped the server with two dual-port Intel 10Gbps Ethernet Network Interface Cards (NICs). The I/O hub interfaces to the NICs (via PCIe) and to additional chipsets on the motherboard that control other peripherals. Other discrete components include the power supply and the fans.

From a power perspective, we consider only the components that consume a non-negligible amount of power: CPUs, memory, NICs, fans, and the motherboard. We use the term "motherboard" as a generic term to include components like the I/O Hub and PCIe bridge. It also includes other system peripherals not directly used in our tests: USB controller, SATA controller, video card, and so forth. The power supply unit (PSU) delivers power to all components using a single 12V DC line to the motherboard, which is in turn responsible for distributing power to all other subsystems (CPUs, fans, *etc.*).

We measure the current absorbed by the system by reading the voltage across 0.05Ω shunt resistors placed in series to the 12V line. This gives us the ability to measure power with good accuracy and sampling frequency and bypassing the PSU.

Workload. Our server runs Linux 2.6.24 and Click [21] with a 10G Ethernet device driver with support for multiple receive and transmit queues. Using multiple queues, a feature available in all modern high speed NICs, we can make sure each packet is handled from reception to transmission by only one core without contention. We use Receive Side Scaling (RSS) [5] on the NIC to define the number of queues and, hence, the number of cores that will process traffic. RSS selects the receive queue of a packet using the result of a hash computed on the 5-tuple of the packet header. This way traffic is evenly spread across queues (and cores). In the rest of the paper, when we refer to a number of cores n we also imply that there are n independent receive queues.

In the following sections, we first focus on the performance and power consumption of a single server operating as an IPv4 router with four 10G ports. Later, in §7, we consider more advanced packet processing applications such as NetFlow, AES-128 encryption and redundancy elimination [10]. For traffic generation we use two additional servers that are set up to generate either synthetic workloads with fixed size packets or trace-driven workloads. The two machines are identical to our router server and each sends packets on two 10Gbps interfaces.

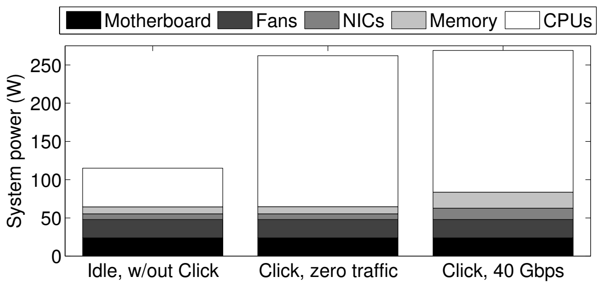


Figure 1: *Breakdown of power consumption across various components when the system is idle and Click is not running (left), with Click running and no traffic (middle), and with Click forwarding 40 Gbps of traffic (right).*

Power characterization. Isolating the power consumption of each component is useful to answer two different questions: (1) where does the power go in a modern server? (2) how does the power consumption vary with load?

The answer to the first question will let us identify which components are the largest consumers while the answer to the second tells us how much can be saved by making that component more energy-proportional.

We measure power in three sets of experiments: i) an idle system without Click running; ii) a system running Click with no input traffic; iii) a system running Click and forwarding 40 Gbps over four interfaces.

Given that we have only one aggregate power measurement for the entire server (plus one dedicated to the NICs), we need to perform several measurements to isolate each component. The system has eight fans, two CPUs and six memory chips. For each scenario we measure the power P with all components and then we physically remove one component (a memory chip, a CPU or a fan) and then measure the power P'. The difference $P - P'$ is then the power consumed by that component.

Figure 1 shows the results of our experiments with IPv4 routing. At peak the system reaches 269 W – 2.3x the idle power consumption of 115 W. With Click running and no traffic the power consumption is 262 W, which is less than 3% lower than the peak power. Several design considerations stem from the results in Figure 1:

The CPUs are clearly the dominant component when it comes to power usage. Even when all cores are idle, they consume almost half of the total idle power, or $\approx$ 25 W each. That energy is mostly used to keep the "uncore" online. At peak, the CPUs reach $\approx$ 92 W each, which is about four times their idle power; this contributes to more than doubling the total power compared to idle.

The other system components contribute little to the over-

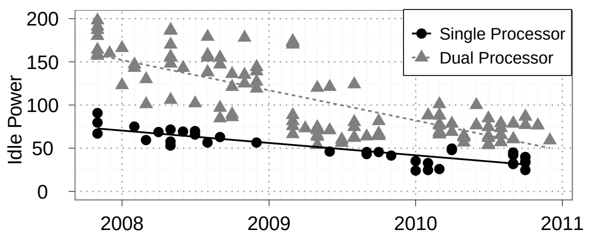


Figure 2: *Server idle power over time (data from [32])*

all power consumption. The motherboard absorbes a constant power (24 W). The memory and NICs exhibit a significant dynamic range but the overall contribution is quite limited (20 W for the six memory chips and 15 W for the four 10 Gbps interfaces).

The system idle power is relatively high at 115 *W.* This is an ongoing concern for system architects and the general trend is towards a reduction in idle power. Figure 2 shows the idle power of a number of systems since 2007 as reported by SpecPower [32]. The plot shows a clear downward trend in idle power, that is expected to continue thanks to the integration of more components into the CPUs and more efficient NICs [9] and memory [7].

Overall, our analysis indicates that to achieve energy efficiency we should focus on controlling the CPUs as they draw the most power and exhibit the largest dynamic range from idle to peak.

Addressing Software Inefficiencies. From Figure 1 we saw that the software stack exhibits very poor power scaling: the total power consumption is almost constant at zero and full load. A pre-requisitive to exploiting power management features of the server is to ensure that the software stack itself is efficient – *i.e.*, doesn't engage in needless work that prevents the creation of idle periods.

In this case, the poor power scaling is caused by Click that disables interrupts and resort to polling the interface to improve packet forwarding performance. An alternative solution is deploy the same mechanisms that can be found in Linux-based systems under the name of Linux NAPI framework. In that framework, interrupts still wake up the CPU and schedule the packet processing thread. That thread then disables the interrupts and switches to polling until the queue is drained or a full batch of packets is processed. It is well understood that this approach will lead to power savings as it gives the CPU an opportunity to transition to a low power state when interrupts are enabled. However, there is no study that show the forwarding latency penalty of using interrupts. To this end, we modified the 10G ethernet driver and Click to operate under the Linux NAPI framework and run power and latency measurements. We call this new codebase NAPI-Click.

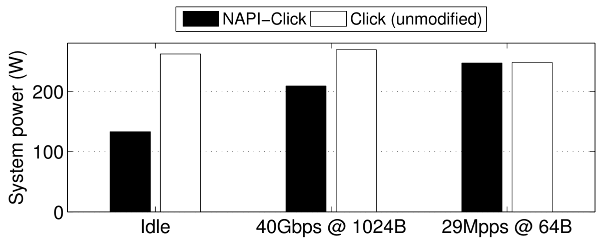


Figure 3: *Power consumption of NAPI-Click vs Click in polling mode.*

Figure 3 shows the power savings with NAPI-Click *vs.* unmodified polling-mode Click for different packet rates. We see that NAPI-Click reduces power consumption by 50% in idle mode and 22% when forwarding 40 Gbps of 1024B packets. On the performance side, enabling interrupts has no impact on the maximum packet rate the router can forward – 29 Mpps with 64B packets.

Regarding latency, we measured a packet forwarding latency across the server of about 7 μs with polling and 12 μs with NAPI-Click. At high load, the latencies of the two methods tend to converge as NAPI-Click degenerates to polling at high load. In the interest of space, we refer the reader to [29] for a more exhaustive description of our setup and analysis of the latency results.

4 Server Support for Power Management

Modern servers provide a variety of power management mechanisms that operate at all levels in the platform. Here, we summarize the controls exposed to the OS.

Core idle states (C-states). Each core in a package can be independently put into one of several *low power* or "idle" states, which are numbered from C1 to Cn. The state C0 refers to a core that is executing instructions and consuming the highest amount of power. Higher numbered states indicate that a successively larger portion of the core is disabled resulting in a lower power draw.

A core enters into a C state either by executing a HALT or MONITOR/MWAIT instruction. C states are exited (to return to the C0 state) when the desired event (interrupt, or write access to the monitored memory range) occurs. With a larger amount of circuitry turned off, higher idle states incur a higher *exit latency* to revert back to the fully operational C0 state. We measure exit latencies for C states in §5.

As an example, the processors we use offer three C states: C1, C3 and C6. A core in C1 is *clock gated*, in C3 the L1 and L2 caches are flushed and turned off while in C6 power distribution into the core is disabled and the core state is saved in the L3 cache.

Processor performance states (P-states). While a processor core is in active C0 state, its power consumption is determined by the voltage and operating frequency, which can be changed with Dynamic Voltage and Frequency Scaling (DVFS). In DVFS, voltage and frequency are scaled in tandem. Modern processors offer a number of frequency and voltage combinations, each of which is termed a P state. P0 is the highest performing state while subsequent P states operate at progressively lower frequencies and voltages. Transitions between P states require a small time to stabilize the frequency and can be applied while the processor is active. P states affect the entire CPU and all cores always run at the same frequency. The processor in our system offers 14 P states with frequencies ranging from 1.6 GHz to 3.3 GHz.

5 Studying the Design Space

We now turn to exploring the design space of power-saving algorithms. A system developer has three knobs by which to control CPU power usage: $i)$ the number of cores allocated to process traffic; $ii)$ the frequency at which the cores run; and $iii)$ the sleep state that cores use when there is no traffic to process.

The challenge is how and when to use each of these knobs to maximize energy savings. We first consider the single core case and then extend our analysis to multiple cores. Our exploration is based on a combination of empirical analysis and simple theoretical models; the latter serving to build intuition for why our empirical results point us in a particular direction.

5.1 Single core case

With just one core to consider our design options boil down to one question: is it more efficient to run the core at a lower frequency for a longer period of time *or* run the core at a (relatively) higher frequency for a shorter period of time and then transition to a sleep state?

To understand which option is superior, we compare two extreme options by which one might process W packets in time T:

- **the "hare"**: the core runs at its maximum frequency, f_{max} and then enters a low-power sleep state (C1 or below). This strategy is sometimes referred to as "race-to-idle" as cores try to *maximize* their idle time.
- **the "tortoise"**: the core runs at the minimum frequency, f_x, required to process the input rate of W/T. I.e., we pick a core frequency such that there is *no* idle time. This is a form of "just-in-time" strategy.

To compare the two strategies we can write the total energy consumption of one core over the period T as:

$$E = P_a(f)T_a(W,f) + P_sT_s + P_iT_i, \qquad (1)$$

where $T = T_a(W,f) + T_s + T_i$. The first term $P_a(f)T_a(W,f)$ accounts for the energy used when actively processing packets. $P_a(f)$ is the active power at frequency f and $T_a(W,f)$ is the time required to process W packets at frequency f. The second term is the energy required to transition in and out of a sleep state, while the third is the energy consumption when idle.
With the *tortoise* strategy, the core runs at a frequency $f = f_x$ such that $T = T_a(W,f_x)$ and no idle time; hence:

$$E_{tortoise} = P_a(f_x)T_a(W,f_x) \qquad (2)$$

With the *hare* strategy, $f = f_{max}$ and hence:

$$E_{hare} = P_a(f_{max})T_a(W,f_{max}) + P_sT_s + P_iT_i, \qquad (3)$$

To facilitate comparison, let's assume (for now) that $T_s = 0$ (instantaneous transitions to sleep states) and $P_i = 0$ (an ideal system with zero idle power). Note that these assumptions greatly favor any *hare*-like strategy.

With these assumption the comparison between the *tortoise* and *hare* boils down to a comparison of their $P_a()T_a()$ terms in Equations 2 and 3. The literature on component-level chip power models tell us that the active power $P_a(f)$ for a component using frequency/voltage scaling grows faster than $O(f)$ but slower than $O(f^3)$.[1] The term $T_a(W,f)$ instead scales as $1/f$ in the best case. Hence, putting these together, we would expect that $P_a(f_{max})T(W,f_{max})$ is always greater than $P_a(f_x)T(W,f_x)$, since $f_{max} \geq f_x$. Hence, despite our very '*hare* friendly' assumptions ($T_s = P_i = 0$), basic chip power models would tell us that it is better to behave like a *tortoise* than a *hare*.

Do experimental results agree with the above reasoning? To answer this, we use the same experimental setup as in §3 with a workload of 64B packets and plot results for our server using between 1 to 12 cores. We show results with more than just one core to ensure we aren't inadvertently missing a trend that arises with multiple cores; in all cases however we only draw comparisons across tests that use the same number of cores.

We first look at how $P_a(f)$ scales with f in practice. Fig. 4 plots the active power consumption, $P_a()$, as a function of frequency. For each data point, we measure power at the maximum input packet rate that cores can sustain at that frequency; this ensures zero idle time and hence that we are indeed measuring only $P_a()$. We see that $P_a()$ does not grow as fast as our model predicted – *e.g.*, halving the frequency leads to a drop of only about 5% in power usage with one core, up to a maximum of 25% with twelve cores. The reason is that, in practice,

[1]This is because power usage in a chip can be modeled as cV^2f where c is a constant that reflects transistor capacitance, V is the voltage and f is the frequency. As frequency increases voltage must also increase – larger currents are needed to switch transistors faster.

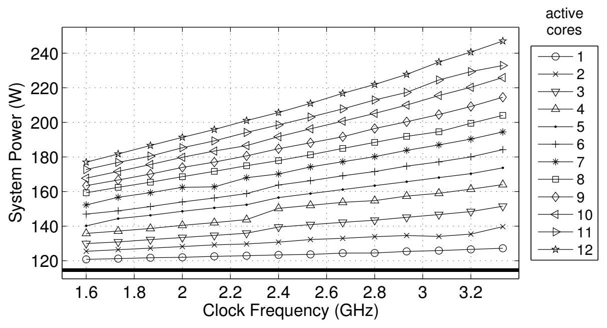


Figure 4: *Power vs. Frequency at maximum sustained rate. The solid black line marks the system idle power.*

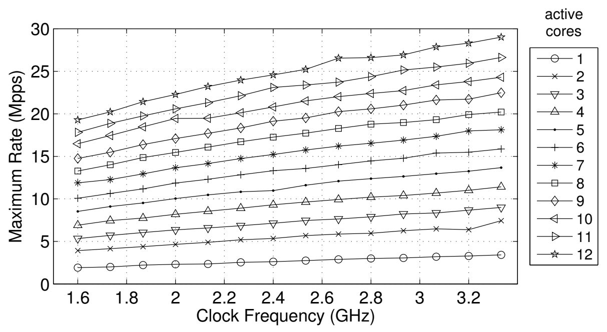


Figure 5: *Packet rate vs. frequency at maximum sustained rate for different number of cores.*

many of the server's components, both within the CPU (*e.g.*, caches and memory controllers) and external to the CPU (*e.g.*, memory and I/O subsystems) are not subject to DVFS leading to lower savings than predicted.

Thus, active power consumption grows much more slowly with frequency than expected. What about $T_a(W, f)$? Since the processing time dictates the forwarding rate a core can sustain, we look at the forwarded packet rate corresponding to each of the data points from Fig. 4 and show the results in Fig. 5. Once again, we see that our model's predictions fall short: doubling the frequency does not double the sustainable packet rate. For example, with one core, doubling the frequency from 1.6 GHz to 3.2 GHz leads to an increase of approx. 70% in the forwarded packet rate (down to 45% with twelve cores). Why is this? Our conjecture is that the perfect $1/f$ scaling of processing time applies only to a CPU-intensive workload. Packet processing however is very memory and I/O intensive and access times for memory and I/O do not scale with frequency. Therefore, as we increase the frequency we arrive at a point where the CPU does its work faster and faster but it is then stalled waiting for memory accesses, leading to a point where increasing the frequency no longer improves productivity.

In summary, we find that, $P_a()$ grows more slowly than expected with frequency but at the same time $T_a()$

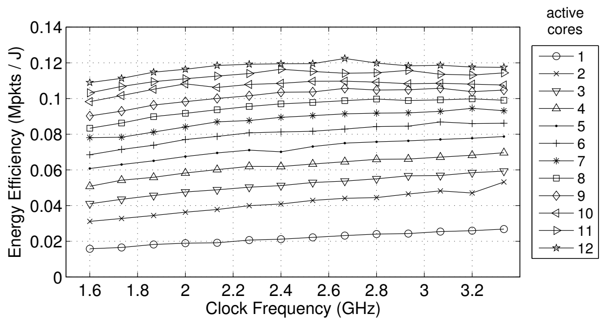


Figure 6: *Processed packets per Joule varying the frequency and the number of cores.*

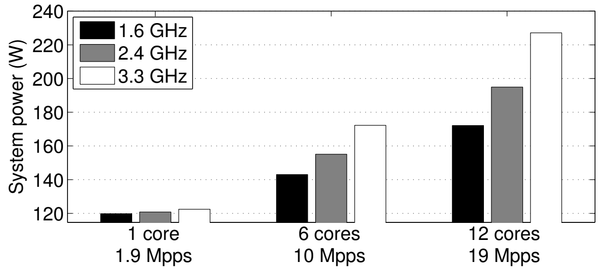


Figure 7: *Power consumption of three frequencies for a constant input rate. At the lowest frequency (black bar) the system has no idle time.*

decreases more slowly than expected. Where does this leave the product $P_a()T_a()$? We cannot directly measure this product (energy) and hence we look at efficiency in terms of packets-forwarded per Joule, obtained by dividing the maximum sustained forwarding rate (from Fig. 5) by the power consumption (Fig. 4). The result is shown in Fig. 6 for increasing frequency. As before, this captures system efficiency while actively processing work (*i.e.*, there's no idle time). We see that, empirically, running at the maximum frequency is marginally more energy efficient in all configurations. That is, if our assumptions of $T_s = P_i = 0$ were to hold, then the *hare* would actually be marginally *more* power-efficient than the *tortoise* – counter to theoretical guidelines.

Of course, our assumptions are not realistic. In particular, we saw earlier (Fig. 3) that P_i is quite high. Hence, the consumption due to the P_i and P_s terms in Eqn. 3) tilts the scales back in favor of the *tortoise*.

The final validation can be seen in Fig. 7 where we plot the total power consumption for a fixed input packet rate at different frequencies. These are the first set of results (in this section) where we include idle and transition times. Based on our analysis from the following section, we make an idle core enter the C1 sleep state. We consider 1, 6 and 12 cores and, for each experiment, we fix the input packet rate to be the maximum rate the system can sustain at a frequency of 1.6 GHz; *i.e.*, at 1.6 GHz,

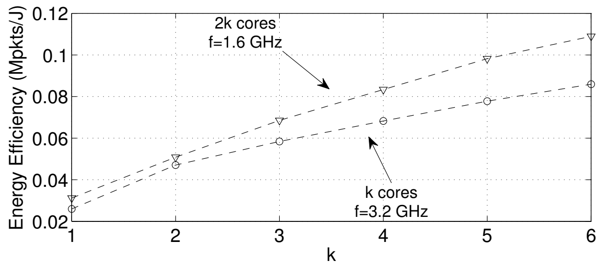


Figure 8: *Comparing processed packets per Joule for k cores at frequency f and nk cores at frequency $\frac{f}{n}$.*

there is no idle time and this corresponds to the *tortoise* strategy; the tests at 2.4 GHz and 3.3 GHz represent the (fast and fastest) *hare* strategy. It is clear from the figure that the configuration with the lowest frequency is always the most energy efficient.

Hence, in summary, the *tortoise* wins in keeping with what the theoretical power model would suggest. However this is *not* because the work is more efficiently processed at low frequency (as the power models would indicate) but because being idle is not sufficiently efficient (*i.e.*, the terms P_iT_i and P_sT_s are quite significant).

5.2 Multiple cores

We now consider the case where we have N cores to process W packets in time T. The results in the previous section show that it is more energy efficient to run an individual core at a low frequency and minimize idle time. Applying this to the multiple cores case would mean dividing the work across multiple cores such that each core runs at a frequency at which it is fully utilized (*i.e.*, no idle time). In doing so, however, we must decide whether to divide the work across fewer cores running at a (relatively) higher frequency *or* more cores at a low frequency. Again, we first look to theoretical models to understand potential trade-offs.

Let us assume that a single core at a frequency f can process exactly W/k packets in time T. Then, our design choice is between:

- **"strength in speed":** use k cores at a frequency f. Each core processes W/k packets, sees no idle time, and hence consumes energy $kP_a(f)T(W/k, f)$;
- **"strength in numbers":** use nk cores at a frequency f/n. Each core now processes W/nk packets, sees no idle time, and hence consumes energy $nkP_a(f/n)T(W/kn, f/n)$.

As before, an idealized model gives $T(W/k, f) = T(W/kn, f/n)$ (since cores at f/n process $1/n$-th the number of packets at $1/n$-th the speed) and hence our comparison is between $kP_a(f)$ and $nkP_a(f/n)$. If the active power $P_a(f)$ grows faster than $O(f)$ (as the theory suggests) then the strength-in-numbers approach is clearly more energy efficient. This points us towards running with the maximum number of cores, each running at the minimum frequency required to sustain the load.

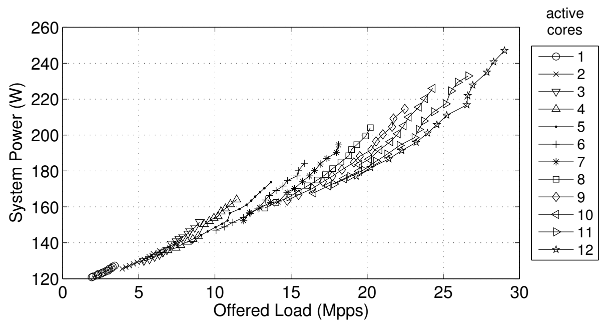


Figure 9: *System power varying number of cores and operating frequency. Each point corresponds to the maximum 64B packet rate that configuration can sustain.*

Looking for empirical validation, we consider again the packets forwarded per Joule but now comparing two sets of configurations: one with k cores running at frequency f and one with nk cores at frequency f/n. Unfortunately, since our hardware offers a frequency range between $[1.6, 3.3]$ GHz, we can only derive data points for $n = 2$ and $k = [1, 6]$ – the corresponding results are shown in Fig. 8. From the figure we see that the empirical results do indeed match the theoretical guidelines.

However, the limited range of frequencies available in practice might raise the question of whether we can expect this guideline to hold in general. That is, what can we expect from m_1 cores at frequency f_1 *vs.* m_2 cores at frequency f_2 where $m_1 < m_2$ and $f_1 > f_2$? To answer this, we run an exhaustive experiment measuring power consumption for all possible combinations of number of cores (m) and core frequency (f). The results are shown in Fig. 9 – for each $\langle m, f\rangle$ combination we measure the power consumption (shown on the Y-axis) and maximum forwarding rate (X-axis) that the m cores can sustain at a frequency f. Thus in all test scenarios, the cores in question are fully utilized (*i.e.*, with no idle times). We see that, for any given packet rate, it is always better to run more cores at a lower frequency.

Applying this strategy under the practical constraints of a real server system however raises one additional problem. A naive interpretation of the strength-in-numbers strategy would be to simply turn on all N available cores and then crank up/down the frequency based on the input rate — *i.e.*, without ever worrying about how many cores to turn on. This would be reasonable if we could tune frequencies at will, starting from close to 0 GHz. However, as mentioned, the frequency range available to us starts at 1.6 GHz and, so far, we've only

sleep state	system power	avg. exit latency
C1	133 W	$< 1\ \mu$s
C3	120 W	60 μs
C6	115 W	87 μs

Table 1: Power consumption and exit latency for different sleep states. See [29] for details on the experimental setup used to measure the above.

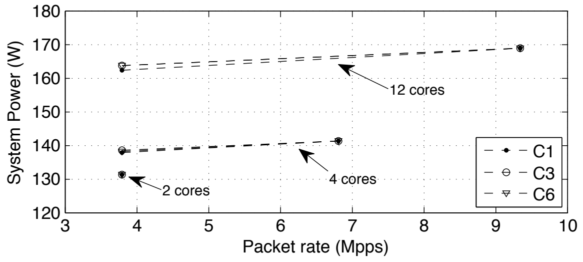


Figure 10: *Comparing the impact of different C states on power consumption. All cores are running at* 1.6 *GHz.*

considered the efficiency of m cores that run *fully utilized* at any frequency. For example, if we consider the 12 core case in Fig. 9 we only see the power consumption for data rates higher than approx. 19Mpps (the max forwarding rate at 1.6 GHz). How would 12 cores fare at lower packet rates?

Before answering this question, we must first decide how to operate cores that are under-utilized *even at their lowest operating frequency*. The only option for such cores is to use sleep states and our question comes down to which of the three sleep states – C1, C3 or C6 – is best. This depends on the power-savings *vs.* exit-latency associated with each C state. We empirically measured the idle power consumption and average transition (a.k.a 'exit') latency for each C state – the results are shown in Table 1. We see that C3 and C6 offer only modest savings, compared to C1, but incur quite large transition times. This suggests that C1 offers the 'sweet spot' in the tradeoff.

To verify this, we measured the power consumption under increasing input packet rates, using a fixed number of cores running at a fixed frequency. We do three set of experiments corresponding to whether an under-utilized core enters C1, C3 or C6. Fig. 10 shows the results for 2, 4, and 12 cores and a frequency of 1.6 GHz. We see that which C-state we use has very little impact on the total power consumption of the system. This is because, even at low packet rates, the packet-interarrival times are low enough (*e.g.*, 1 Mpps implies 1 packet every $1\mu s$) that cores have few opportunities to transition to C3 or C6. In summary, given its low transition time, C1 is the best choice for an under-utilized core.

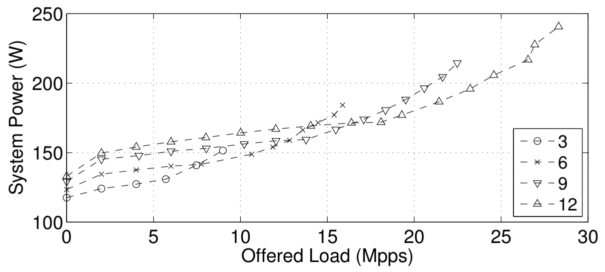


Figure 11: System power consumption varying the input data rate. Cores transition to C1 when underutilized.

Now that we know what an under-utilized core should do, we extend the results from Fig. 9 by lowering the input data rates to consider the case of under-utilized cores. For clarity, in Fig. 11 we plot only a subset of the data points. We see that, for any packet rate, the appropriate strategy is *not* to run with the maximum cores at the lowest frequency but rather to run with the maximum cores that can be kept *fully utilized.* For example, at 5 Mpps, the configuration with 3 active cores saves in excess of 30 W compared to the 12 cores configuration.

In summary, our study reveals three key guidelines that maximize power savings:

1. **strength in numbers:** the aggregate input workload should be equally split between the maximum number of cores that can be kept fully utilized.

2. **act like a tortoise:** each core should run at the lowest possible frequency required to keep up with its input workload.

3. **take quick-n-light naps:** if a single core running at its lowest possible frequency is under-utilized, it should enter the lightest C1 sleep state.

Following the above guidelines leads to the lower envelope of the curves in Fig. 9 which represents the optimal power-consumption at any given packet rate. In the following section, we describe how we combine these guidelines into a practical algorithm.

6 Implementation

Our empirical results tell us that a strategy that yields the maximum power saving is one that tracks the lower envelope of the curves in Figure 9. A simple implementation of that strategy could be to use a lookup table that, for any input data rate, returns the optimal configuration. However, such an approach has two limitations. First, perfect knowledge of instantaneous data rate is required. Second, any change in the hardware would require comprehensive benchmarking to recompute the entire table.

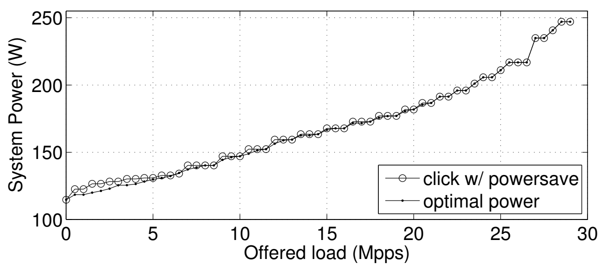


Figure 12: *Power usage as a function of packet rate for our power saving algorithm and an optimal strategy.*

An alternative is to devise an algorithm that tries to approximate the lower envelope of the curves with no explicit knowledge of those curves. An approach that maps quite well to the guidelines is the following iterative algorithm (that we will call "PowerSave"):

- Start with only one core active at the minimum frequency. All other cores are in C6 deep power down state ("inactive").
- As the load increases wake up one more core and split the traffic evenly among all active cores. If all cores are active, increase the frequency of all cores to keep up with the input data rate.
- If traffic load decreases, first lower the frequency of all cores and then start turning off one core at a time consolidating traffic onto the remaining active cores.

To understand how well this algorithm could approximate the optimal power curve we emulate its behavior by using the power measurement used to plot Figure 9. In Figure 12, we show the power usage for an optimal algorithm, that follows the lower envelope of Figure 9, and for our PowerSave algorithm. The approximation closely tracks the optimal algorithm. At very low rate our algorithm chooses to turn two cores on much earlier than the optimal solution would. Even so, the additional power consumption is very small (below 5 W) given that the two cores always run at the lowest frequency.

Turning this algorithm into an actual implementation requires two mechanisms: i) a way of determining whether the current configuration is the minimum required to sustain the input rate, and ii), a way of diverting traffic across more cores or consolidating traffic onto fewer cores. The remainder of this section describes our implementation of the two mechanisms.

6.1 Online adaptation algorithm

The goal of the adaptation algorithm is to determine if the current configuration is the minimum configuration that can sustain the input traffic rate. The algorithm needs to be able to quickly adapt to changes in the traffic load and do so with as little overhead as possible.

A good indicator of whether the traffic load is too high or low for the current configuration is the number of packets in the NIC's queues. Obtaining the queue length from the NICs incurs very little overhead as that information is kept in one of the NIC's memory-mapped registers. If the queues are empty – and stay empty for a while – we can assume that too many cores are assigned to process the incoming packets. If the queues are filling up instead, we can assume that more cores (or a higher frequency) are needed to process the incoming packets.

This load estimation is run as part of the interrupt handler to allow for a quick response to rate changes. Only one core per interface needs to run this estimator to decide when to wake up or put to sleep the other cores. We set two queue thresholds (for hysteresis) and when the number of packets is below (above) a threshold for a given number of samples we turn off (turn on) one core The pseudocode of the algorithm is the following:

```
for (;;) {
   // get a batch of packets and queue length
   qlen, batch = input(batch_len);

   // now check current queue length
   if (qlen > q_h) {
      // the queue is above the high threshold.
      c_l = 0; c_h++;
      if (c_h > c_up) {
         c_h = 0;
         <add one core, if all are on, raise frequency>
      }
   } else if (qlen < q_l) {
      // the queue is below the low threshold
      c_l++; c_h = 0;
      if (c_l > c_down) {
         c_l = 0;
         <decrease frequency, if at min remove one core>
      }
   } else { c_h = c_l = 0; }
   // process a full batch of packets
   process(batch);

   if (qlen == 0) halt();
}
```

The parameters q_h and q_l are used to set the target queue occupancy for each receive queue. Setting low queue thresholds leads to smaller queueing delays at a cost of higher power usage. The choice of the most appropriate thresholds is best left to network operators: they can trade average packet latency for power usage. In our experiments we set $q_h = 32$ and $q_l = 4$ for a target queueing delay of ≈ 10 μs – assuming a core can forward at 3 Mpps.

The variables c_h and c_l keep track of how many times the queue size is continuously above or below the two thresholds. If the queue is above the q_h threshold for a sufficient time, the system is under heavy load and we add one more core or increases the frequency. Conversely, if we are under the q_l threshold for a sufficient time, we decrease the frequency or turn off one core. Note that every time we change the operating point, the counter is reset to give the system sufficient time to react to the change. After adjusting the operating level, we

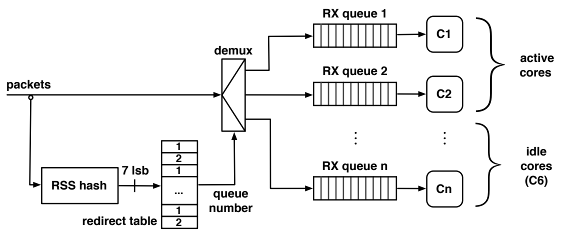


Figure 13: *NIC queues and redirect table. Hashes are computed on the 5-tuple of each packet, and used to address one of the 128 entries of the redirect table.*

process a batch of packets, and either repeat the loop, or halt (thus moving to state C1) if there are no more packets to be processed. In case of a halt, the next interrupt will resume processing with no further delays other than those related to interrupt generation and dispatching.

6.2 Assigning queues to cores dynamically

Our algorithm requires that the NIC be able to split the incoming traffic evenly between a variable set of cores. For this we use the Receive Side Scaling (RSS) [5] mechanism that implement the support for multiple input and output queues in modern NICs.

For each incoming packet, the NIC computes a hash function on the five-tuple (source and destination IP, source and destination port, and protocol number) of the packets. The 7 least significant bits of the hash are used as an index into a *redirect table*. This table holds 128 entries that contain the number of the queue to be used for packets with that hash, as shown in Figure 13. Each queue has a dedicated interrupt and is assigned to one core. The interrupt is raised on packet reception and routed to the associated core.

The number of queues on a NIC can only be set at startup. Any change in the number requires a reset of the card. For this reason we statically assign queues to cores as we start Click. To make sure inactive cores do not receive traffic, we modify the redirect table by mapping entries to queues that have been assigned to active cores.The entries are kept well balanced so that traffic is spread evenly across active cores. The table is only 128 bytes long, so modifications are relatively fast and cheap.

If a core does not receive traffic on its queue it remains in deep power down. When additional cores are needed to process traffic, we reconfigure the redirect table to include the additional queue. The update to the redirect table has almost immediate effect; this is important because it immediately reduces the load on the active cores. As soon as the first packets arrives to the new queue, the interrupt wakes up the corresponding core, which eventually (after the exit latency) starts processing the traffic.

Taking out a core is equally simple: we reprogram the redirect table so that no new packets will reach the extra core. Also, we instruct the core to enter a C6 state on `halt()` instead of C1. Eventually, the queue will become empty, and the core will enter C6 from which it does not exit until the queue is enabled again.

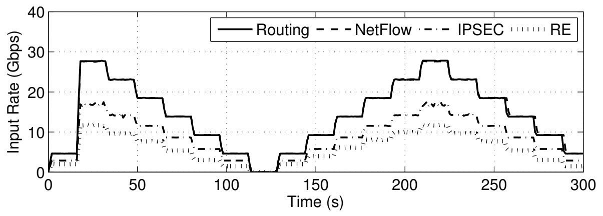


Figure 14: *Input traffic profile for different applications.*

7 Evaluation

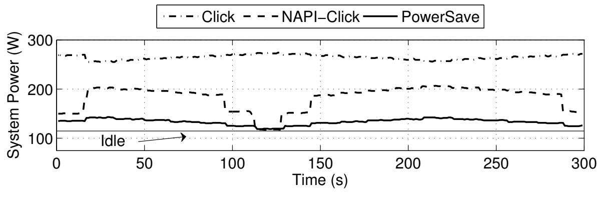
To evaluate the performance of our prototype router in a realistic setting we generate traffic using two packet traces: the "Abilene-I" trace collected on the Abilene network [4] and a one day long trace from the "2009-M57 Patents" dataset [3] collected in a small-enterprise 1Gbps network. The former contains only packet headers while the latter includes packet payloads as well.

Our generator software is composed of many traffic sources, each reading from a copy of the original traces. By mixing multiple traffic sources we can generate high bit rates as well as introduce burstiness and sudden spikes in the traffic load. This helps in testing the responsiveness and stability of our power saving algorithm.

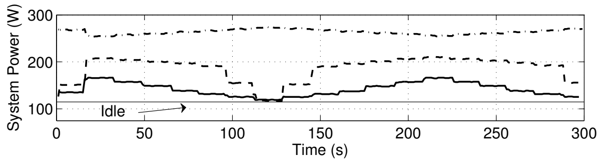
We are interested in evaluating the performance of our system with a broad range of applications. To this end, we run experiments with the following packet processing applications: IPv4 routing, a Netflow-like monitoring application that maintains per-flow state, an IPSEC implementation that encrypts every packet using AES-128 and the Redundancy Elimination implementation from [10] which removes duplicate content from the traffic. Note that we use all applications *as-is*, without modifications from their original Click-based implementations.

This set of applications is quite representative of typical networking applications. Routing is a state-less application where each packet can be processed independently of the other. Netflow is stateful and requires to create and maintain a large amount of per-flow state. IPSEC is very CPU intensive but stateless. Finally, Redundancy Elimination is both stateful (keeps information about previously observed packets) and CPU intensive (to find duplicate content in packet payloads).

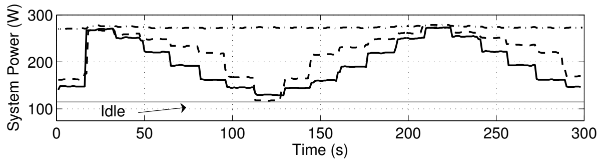
We generate similar input traffic profiles for all applications. Figure 14 shows the traffic load over time for the different applications. We tune the traffic profiles so that the average utilization of the router over the duration of the experiment is around 35%. This results in



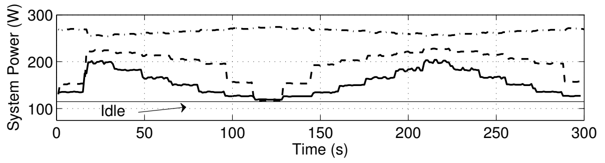
(a) Routing



(b) NetFlow



(c) IPSEC



(d) Redundancy Elimination

Figure 15: *Power usage with (a) IPv4 routing, (b) NetFlow, (c) IPSEC, (d) Redundancy Elimination.*

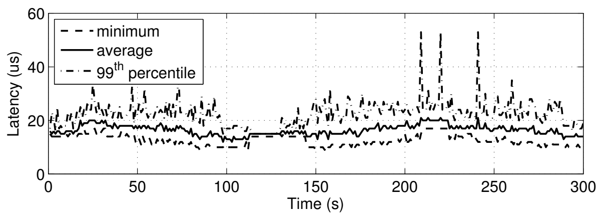


Figure 16: *Average, minimum and 99th percentile of the packet forwarding latency when using Click with the power save algorithm.*

different bit rates for different applications as they differ in per-packet processing cost (with Redundancy Elimination being the most demanding). The load is evenly distributed across the four interfaces and the routing table is uniform so that no output interface has to forward more than the 10 Gbps it can handle.

We first focus on the power consumption of the various applications and then measure the impact of *PowerSave* on more traditional performance metrics such as latency, drops and packet reordering.

Power consumption. Figure 15[a-d] shows the power consumption over time for the four applications under study. In the figures we compare the power consumption of the default Click implementation, the NAPI-Click and Click with PowerSave. As expected, unmodified Click fares the worst with a power consumption hovering around 270 W for all applications at all traffic rates. The power usage with NAPI-Click and PowerSave instead tracks, to varying degree, the input traffic rate.

PowerSave is consistently the most energy efficient at all rates for all applications. The advantage of PowerSave is more prominent when the load is fairly low $(10-20\%)$ as that is when consolidating the work across cores yields the maximum benefits. Over the duration of the experiments, PowerSave yields an overall energy saving between 12% and 25% compared to NAPI-Click and 27-50% compared to unmodified Click.

Traditional performance metrics. We turn our attention to other metrics beyond power consumption, namely: packet loss, latency and reordering. Our algorithm may introduce loss or high latency by turning on cores too slowly in the face of varying traffic demands. Reordering may be introduced when packets belonging to the same flow are redirected to a different core.

Regarding packet losses, the PowerSave algorithm reacts quickly enough to avoid packet drops. Indeed, no packet losses were observed in any of our experiments.

We also measured the latency of packets traversing the router. Figure 16 plots the average, minimum and 99th percentile over each 1s interval – we collect a latency sample every 1ms. In the interest of space we only plot results for one PowerSave experiment with IPv4 Routing. Other experiments show similar behavior [29].

The average latency hovers in the $15 - 20$ μs range. The same experiments with Click unmodified yield an average latency of 11 μs. The 99th percentile of the latency is up to 4 times the average latency – it peaks at 55 μs. Some of the peaks are unavoidable since waking up a core on from C6 may introduce a latency of up to 85 μs. However, that is limited to the first packet that reaches the queue handled by that core.

As a last performance metric we also measured packet reordering. In our system, reordering can only occur when traffic is diverted to a new queue. Hence, two back to back packets, A and B, belonging to the same flow may be split across two queues and incur very different latency. Given that new queues are activated only when the current ones start to fill up, it is possible to have packet A at the back of one queue while packet B is first in line to be served on the new queue. On the other hand, packets in the newly activated queue will not be processed until the core assigned to it exits a C6 state. Given that in our setting the adaptation algorithm aims for $\approx$ 10 μs of queueing delay, it is quite likely that packet A will be served while the new core is still exiting C6. We confirmed this conjecture in our experiments

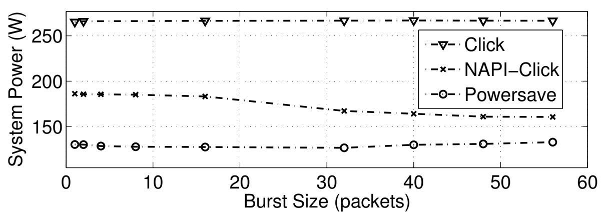


Figure 17: *System power varying the burstiness of the input traffic. The average traffic rate is constant at* 2.5 *Gbps per interface*

where we have measured zero packets being reordered.

Impact of traffic burstiness. Our results so far were based on input traffic whose fine-grained "burstiness" derived from the natural variability associated with a software-based traffic source and multiplexing traffic from multiple such sources. To study the impact of fine-grained burstiness in a more controlled manner, we modified the traffic sources to generate traffic following an on/off pattern. During an "on" period the traffic source generates a burst of back-to-back packets at the maximum rate. We control the burstiness by setting the length of the "on" period (*i.e.*, the number of back-to-back packets) while keeping the average packet rate constant.

Fig. 17 shows the power usage of our system as we vary the burstiness of the sources (a value of 1 corresponds to uniform traffic). The average traffic rate is around 2.5 Gbps per interface. In the interest of space we only plot the results for IPv4 Routing. Other applications exhibit a similar behavior [29]. As we can see the power usage varies very little as we increase the burstiness. This shows that the controller is stable enough to make sure short-term burstiness does not impact its performance. As expected, the power usage with unmodified Click is flat while NAPI-Click benefits from traffic burstiness as interrupts are spaced out more with burstier traffic. However, for all levels of burstiness, PowerSave is clearly the most energy efficient. We measured also latency and packet loss and saw similar results where burstiness has little or no impact. We refer the reader to [29] for a detailed analysis of the latency results.

8 Conclusion

We tackle the problem of improving the power efficiency of network routers without compromising their performance. We studied the power-*vs*-performance tradeoffs in the context of commodity server hardware and derived three guiding principles on how to combine different power management options. Based on these guidelines we build a prototype Click-based software router whose power consumption grows in proportion to the offered load, using between 50% and 100% of the original power, without compromising on peak performance.

References

[1] BlueCoat Proxy. `http://www.bluecoat.com`.

[2] Cisco Green Research Symposium. `http://goo.gl/MOUNl`.

[3] "M57 Patents" Dataset. `http://goo.gl/JGgw4`.

[4] NLANR: Internet Measurement and Analysis. `http://moat.nlanr.net`.

[5] Receive-Side Scaling Enhancements in Windows Server 2008. `http://goo.gl/hsiU9`.

[6] Riverbed Steelhead:. `http://www.riverbed.com`.

[7] SAMSUNG Develops Industry's First DDR4DRAM, Using 30nm Class Technology. `http://goo.gl/8aws`.

[8] Vyatta Router and Firewall. `http://www.vyatta.com`.

[9] Media Access Control Parameters, Physical Layers and Management Parameters for Energy-Efficient Ethernet. IEEE 802.3az Standard, 2010.

[10] A. Anand et al. Packet caches on routers-the implications of universal redundant traffic elimination. In *ACM Sigcomm*, 2008.

[11] G. Ananthanarayanan and R. Katz. Greening the switch. In *HotPower*, 2008.

[12] M. Anwer et al. Switchblade: A platform for rapid deployment of network protocols on programmable hardware. 2010.

[13] J. S. Chase et al. Managing energy and server resources in hosting centers. In *SOSP*, 2001.

[14] M. Dobrescu et al. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *ACM SOSP*, 2009.

[15] W. Fisher et al. Greening Backbone Networks: Reducing Energy Consumption by Shutting Off Cables in Bundled Links. In *ACM Green Networking Workshop*, 2010.

[16] M. Gupta and S. Singh. Greening of the Internet. In *ACM Sigcomm*, 2003.

[17] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated Software Router. In *ACM Sigcomm*, 2010.

[18] B. Heller et al. ElasticTree: Saving Energy in Data Center Networks. In *ACM NSDI*, 2010.

[19] V. Janapa Reddi et al. Web search using mobile cores: quantifying and mitigating the price of efficiency. In *ISCA*, 2010.

[20] C. Joseph et al. Power awareness in network design and routing. In *IEEE Infocom*, 2008.

[21] E. Kohler, , et al. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[22] K. Lim et al. Understanding and designing new server architectures for emerging warehouse-computing. In *ISCA*, 2008.

[23] G. Lu et al. Serverswitch: A programmable and high performance platform for data center networks. In *NSDI*, 2011.

[24] P. Mahadevan et al. A Power Benchmarking Framework for Network Devices. In *IFIP Networking*, 2009.

[25] G. Maier et al. On Dominant Characteristics of Residential Broadband Internet Traffic. In *ACM IMC*, 2009.

[26] D. Meisner et al. Power management of online data-intensive services. In *ISCA, June*, 2011.

[27] D. Meisner, B. T. Gold, and T. F. Wenisch. Powernap: eliminating server idle power. In *ASPLOS*, 2009.

[28] S. Nedevschi et al. Reducing Network Energy Consumption via Sleeping and Rate-Adaptation. In *Proc. of NSDI*, 2008.

[29] L. Niccolini et al. Building a Power-Proportional Router (extended version). Technical report, UCB, 2011.

[30] P. Reviriego et al. Using coordinated transmission with energy efficient ethernet. In *Networking*, May 2011.

[31] V. Sekar et al. The Middlebox Manifesto: Enabling Innovation in Middlebox Deployment. In *ACM HotNets-X*, 2011.

[32] SPECpower ssj2008 results. `http://goo.gl/h6Z2F`.

[33] N. Tolia et al. Delivering energy proportionality with non energy-proportional systems. In *HotPower*, 2008.

[34] D. Tsirogiannis, S. Harizopoulos, and M. A. Shah. Analyzing the energy efficiency of a database server. In *SIGMOD*, 2010.