Rethinking Polling Efficiency in Service Core Network Stacks

Matheus Stolet
mstolet@mpi-sws.org
Max Planck Institute for Software Systems
Germany

Simon Peter
simpeter@cs.washington.edu
University of Washington
USA

Antoine Kaufmann
antoinek@mpi-sws.org
Max Planck Institute for Software Systems
Germany

Abstract

Idle network service cores are treated as wasted compute. This assumption motivates increasingly sophisticated mechanisms that reclaim idle cores at microsecond timescales. We argue that this view no longer matches modern server hardware. On contemporary multicore processors, active cores compete for a shared package level power and thermal budget. Once that budget becomes the limiting resource, an idle core that waits efficiently returns compute capacity that hardware can redistribute to productive work. Measurements on a recent AMD EPYC processor show how waiting strategy, processor topology, and idle duration determine this tradeoff. Our results suggest that reclaiming idle cores often yields less benefit than commonly assumed while introducing substantial scheduling complexity. We propose a budget centric view of service core systems in which power, rather than core occupancy, becomes the fundamental resource and waiting policy becomes a first class systems design choice.

1 Introduction

High-performance network stacks increasingly use *service-core architectures*: packet processing runs on dedicated cores as a shared service, and applications communicate via shared-memory queues rather than system calls [20, 27, 31, 34, 35]. Dedicating cores keeps applications and the stack from competing for caches, branch predictors, and other core-local state; parallelizes processing; removes context switches and privilege transitions from the data path; and lets one shared instance multiplex many applications.

But dedicated service cores are rarely busy every cycle. Load fluctuates at microsecond timescales, and demand does not align with integer multiples of a core. The community treats these idle cycles as waste to be eliminated, and has built sophisticated machinery to do so, such as core reallocation at microsecond granularity, dedicated scheduler cores, and elaborate handoff protocols [10, 27, 32]. This machinery follows a seemingly obvious mental model: a core-cycle spent idle is a core-cycle of computation lost. We argue that this model is wrong on modern servers, leading to systems more complex and *less* efficient than doing nothing.

Large multicore processors face fixed power and thermal budgets [7, 11]. For instance, an AMD EPYC 9655 guarantees 2.6 GHz across its 96 cores but boosts a single core to 4.5 GHz. This gap shows that N cores do not deliver N cores' worth of peak compute. Furthermore, this is not a corner case. Production fleets oversubscribe power, so saturated machines routinely run against their caps [9, 21, 23, 38]. On our EPYC 9655, a cache-resident parallel workload running on all 96 cores achieves only 47% of the per-core throughput it achieves alone. Thus cores are not independent units of compute and performance is based on a shared budget.

Below saturation, reclaiming a service core offers little. If an application is not fully utilizing its current cores, reallocation cannot improve performance. If it is saturating them, the scenario that motivates microsecond core reallocation, the processor is constrained by its power budget. Elastic allocation therefore only helps workloads for which this underlying accounting no longer holds. In that regime, the arithmetic of idleness inverts and an idle core does not squander its capacity. Instead the processor's firmware redistributes it in microseconds with no software in the loop.

We quantitatively characterise this regime and measure how waiting mechanism, idle patterns, placement, and reallocation determine how much budget idle service cores return. Our insights lead us to propose a replacement mental model: **the processor is a fixed compute budget, not a collection of cores.** Below the power cap, cores behave classically; above it, they compete for a shared budget, and one must consider how much budget they consume instead of how many cores a service holds. How a core *waits* becomes as important as whether it is allocated: hardware waiting mechanisms—`PAUSE`, user-level monitor/wait, halt-class deep idle—return progressively more budget at higher wakeup latency without giving up the core. Blocking, by contrast, releases the core to a scheduler, paying wakeups, cache refills, and engineering complexity to redistribute a budget the hardware already redistributes for free. This cost is worthwhile only once idle periods amortise it, a break-even point that power capping shifts past microsecond-granularity.

We distil four lessons for service core architectures, and related multi-core systems. (1) Waiting returns power and

thermal headroom, so idleness is a transfer, not a loss. (2) The compute budget is distributed hierarchically across the CPU, so task placement decides who shares it. (3) Waiting trades wakeup latency for returned budget, so waiting policies should adapt to the workload. (4) Core reallocation is worthwhile only after blocking overheads are amortised, so our systems should stop paying the cost of complexity to eliminate idle cycles that were never wasted to begin with.

2 Background

Modern network stacks and processors have evolved considerably over the past decade, changing the trade-offs around dedicated polling. Many of the assumptions that shaped earlier network stack designs no longer hold on modern servers.

2.1 Managing Service Cores

Service cores isolate operating system services on dedicated cores so applications can run with less interference from shared execution. That separation helps latency sensitive systems, but creates challenges for allocation. Service demand varies over time, and service cores are often provisioned for peak load, so those cores sit idle when demand dips.

Microsecond core reallocation. Recent work treats idle service cores as stranded resources that should be reclaimed. These systems dynamically reallocate cores between applications and services as demand changes, often at microsecond timescales, to recover otherwise idle polling cycles [10, 28, 32]. This approach has driven increasingly sophisticated scheduling mechanisms, but it also relies on rapid coordination, migration, and placement decisions whose overhead must be amortized over the reclaimed work.

Low-overhead core handoffs. A complementary line of work reduces the latency of handing a core new work. These systems employ low-overhead preemption, yielding, and interrupt delivery to shorten the critical path between an idle core and useful execution [2, 16, 17, 24]. These mechanisms reduce the cost of scheduling decisions, but they do not assess if reclaiming idle service cores is beneficial.

Waiting strategies. Service cores use different waiting strategies for short-term idling. Busy polling does not reduce core power consumption but resumes immediately. `PAUSE` spin loop hints let the core reduce power and with multi-threading enable the peer thread to use more cycles, while preserving 10–100 ns-scale wakeups [15, 22]. `UMWAIT` and `MWAIT` save more power by entering deeper idle states [15] but with increased wakeup latency. Deeper power-saving states, such as `HLT` require slow, interrupt-driven wakeups. These strategies trade-off energy and multi-threading efficiency against wakeup latency and form a waiting spectrum.

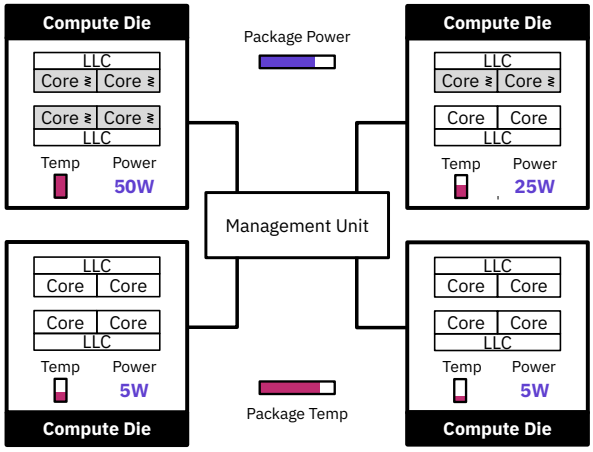

Figure 1: Chiplet-based CPU with four compute dies and four cores per die. Management unit controls power and frequency across cores by monitoring temperature, voltage, and performance counters in individual chiplets and keeps the package under its limits.

2.2 Power and Thermal Management in Large Multicore Machines

The large multicore processors on which these services run operate within a package-level power limit. Hardware and firmware continuously allocate this finite limit across active cores, reducing their frequency when concurrent activity exceeds the package's power or thermal envelope. Thus, activating another core reduces the frequency and the performance of work running elsewhere on the socket [7, 11].

Processor performance depends on package activity. In this paper we use the AMD EPYC 9655 to illustrate this constraint. The processor integrates 96 cores within a 400 W default TDP and advertises a maximum single-core boost frequency of 4.5 GHz, but an all-core boost frequency of 4.1 GHz [1]. The gap between maximum single-core and all-core frequency exemplifies that the frequency available to each core depends on activity across the package. Consequently, cores devoted to polling can consume power headroom even when they perform no useful network processing.

Topology influences power and thermal management. The EPYC 9655 organizes its cores into twelve eight-core compute dies. Because each compute die is physically distinct, its power density, temperature, and available frequency differs as load is distributed across the package [26, 33]. A polling thread may therefore interfere differently with application threads depending on whether they share a core, a die, or the socket. Task placement is consequently a power-management and cache-locality decision. Figure 1 illustrates power and thermal management in a modern chiplet CPU.

3 The Fixed-Budget Model

The conventional view of multicore processors treats each core as an independent unit of compute. Under this model, an idle polling thread wastes compute because its cycles could instead execute application instructions. This intuition holds while the processor has unused power and thermal headroom. In this section, we evaluate application performance under constrained thermal and power regimes and show that this intuition breaks when the processor reaches its package limits. From these results, we propose lessons for service core network stacks (L1-L4) and a new mental model where the processor behaves like a unit under a fixed compute budget, rather than a collection of independent cores.

3.1 Experimental Setup

The benchmarks are run on a machine with Linux kernel version 6.12.86 and a single-socket AMD EPYC 9655 processor with 12 dies, 8 physical cores per die, 96 physical cores in total, and 192 threads with SMT enabled. We sample package power using Linux RAPL counters and per-workload CPU frequency using per-core Linux frequency counters.

Benchmarks. Across experiments, we use a matrix multiplication benchmark as the compute baseline and several polling microbenchmarks as waiting baselines. The matrix multiplication workload operates on 16×16 matrices with either scalar or 512-bit vector AVX instructions. The polling baseline runs in different modes that represent different waiting strategies (`Busy`, `Pause`, `Block`) and pseudo-packet processing tasks (`Work`). `Busy` repeatedly spins on a shared value, `Pause` performs the same spin loop but inserts the `PAUSE` instruction in the poll loop, and `Block` sleeps between polls. `Work` combines polling with the processing of a structure that mimics accesses to IP/TCP headers and checksum computation. All workloads are pinned to explicit cores and averaged over three 60-second runs, unless otherwise stated.

3.2 The Power and Thermal Knee

Core count is a poor proxy for compute capacity once a processor approaches its package limits. To expose this effect, we run a matrix multiplication workload on an increasing number of physical cores. Figure 2 reports the throughput for scalar and AVX baselines. Initially, each additional core contributes substantial useful work. The curve then reaches a knee in which throughput is nearly flat while more cores are activated, before increasing again at a lower rate.

Package-level power and thermal limits reduce per-core performance. This knee is a consequence of package-level power and thermal management. Activating another core increases potential parallelism, but also forces the processor to divide finite electrical and thermal headroom. Dynamic voltage and frequency scaling responds by reducing

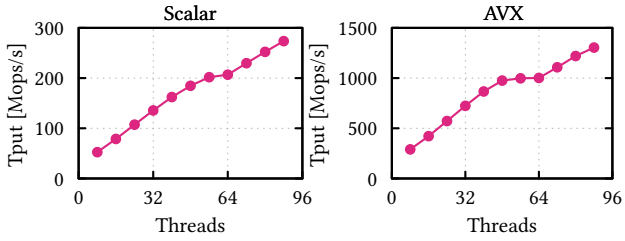

Figure 2: Aggregate compute throughput as the workload increases the number of cores. Throughput briefly flattens near the package's power and thermal knee, after which each additional core contributes less work.

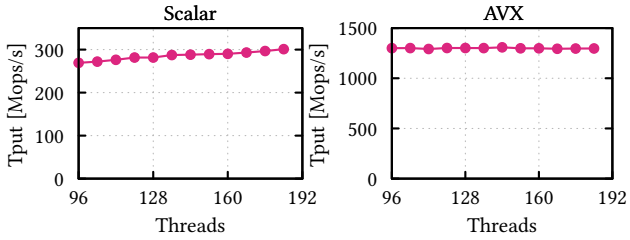

Figure 3: Aggregate compute throughput flattens once the workload scales beyond physical cores to SMT.

core frequency. Around the knee, where the processor starts to more aggressively downclock, the loss on already-active cores can nearly cancel the work provided by a newly activated core. Beyond it, throughput still grows, but the incremental contribution of each core is smaller. Before the knee, each additional core contributes an average of 4.24 million operations per second for scalar matrix multiplication. Beyond the knee plateau, that falls to 2.79 million operations per second. For AVX, the contribution drops from 22.6 to 12.6 million operations per second per additional core.

SMT threads provide little additional compute. Figure 3 shows that once the processor exhausts its physical cores, SMT threads contribute even less additional compute. For compute-intensive workloads, such as AVX, aggregate throughput nearly plateaus despite activating more hardware threads, while scalar workloads see minimal throughput increase. Here adding cores provides diminishing returns because they compete for the same package budget.

Fixed-Budget Model. Below the processor package limit, additional active cores increase compute. Above those limits, additional cores yield marginal performance gains. Activity on one core reduces the budget available to others. Idling offers that budget back for redistribution.

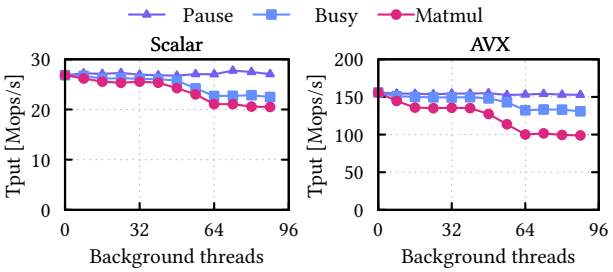

Figure 4: Foreground matrix multiplication throughput as increasing core counts perform background work (`Matmul`, `Busy`, or `Pause`). `Pause` preserves foreground capacity, whereas compute and busy polling consume the shared package budget.

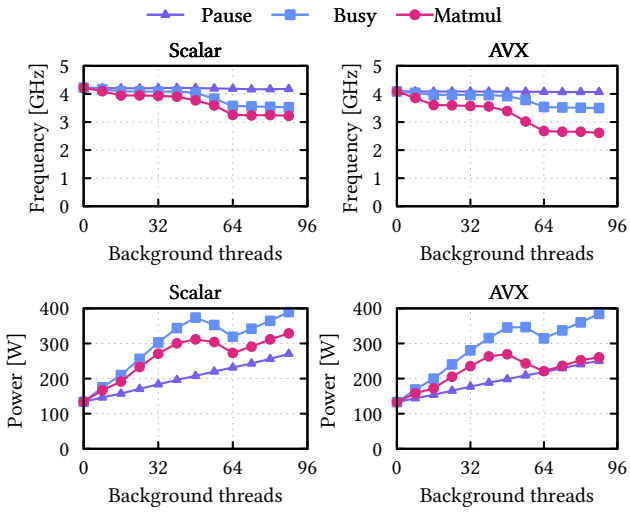

Figure 5: Foreground core frequency (top) and package power (bottom) as cores are assigned to background work (`Matmul`, `Busy`, or `Pause`). `Pause` polling preserves foreground frequency and uses less package power than busy polling.

3.3 The Waiting Spectrum

In the fixed budget regime, a waiting thread returns compute budget that the processor can automatically redistribute to productive work by entering hardware idle states. We isolate this redistribution with an eight-thread matrix multiplication foreground workload. We then place an increasing number of background threads on otherwise unused physical cores. The background either performs matrix multiplication or polls queues, akin to a low-latency network stack, with different waiting strategies. Figure 4 reports foreground throughput, while Figure 5 reports core frequency and total power.

Waiting strategy determines performance of foreground application. The results highlight the spectrum of waiting costs. Scalar compute-heavy background work drives the foreground from 26.8 to 20.5 Mop/s, and busy polling reduces it to 22.5 Mop/s. In both cases, package activity consumes headroom and the processor downclocks the foreground. By contrast, `PAUSE` polling retains foreground throughput and frequency while using substantially less power than `Busy`.

Compute-intensive AVX workloads magnify cost of budget contention. AVX instructions amplify the effects of the fixed budget, as the chip reaches its limits. Throughput drops from 155.9 to 130.9 Mop/s when running background busy polling and further reduces to 98.9 Mop/s with vectorized matrix multiplication. `Pause` polling yet again keeps performance of the foreground application constant as more cores are added.

L1: Idleness is a transfer, not a loss. An idle service core does not necessarily waste compute. Efficient waiting returns power and thermal headroom that hardware redistributes to productive work.

3.4 Placement Shapes Budget Sharing

We investigate the effect of CPU topology on the fixed budget (Figure 6). Each run uses 16 threads for scalar matrix multiplication and 16 threads for polling. We place polling threads in three topologies: applications share dies while using different cores (Die), share the same socket but use disjoint dies (Socket), or share a physical core with sibling hyperthreads (SMT). For each placement, we measure how `Busy` and `Pause` affect matrix multiplication throughput, frequency, and package power.

Waiting strategy reduces interference on SMT cores. When two compute-intensive applications share a physical core, performance degrades when they compete for the same execution units. The correct waiting strategy mitigates the slowdown from polling by freeing the CPU power budget during idle periods. With SMT placement, `Pause` reduces package power by 18.7% compared to `Busy` and improves matrix multiplication throughput by 19.2%.

Thread placement can dissipate hotspots. Individual chiplets can become hotspots and DVFS can downclock cores in that chiplet. The right waiting strategy can dissipate these hotspots so applications can run at higher clock frequencies. For example, when `Pause` polling is co-located on the same die as matrix multiplication, the reduced power consumption on that chiplet allows the cores running matrix multiplication to boost from 4.28 GHz with `Busy` polling to 4.47 GHz. This yields a modest 2.8% performance improvement, but

Figure 6: Per-thread scalar matrix multiplication throughput, matrix multiplication core frequency, and package power when co-located on the same CPU with 16 polling threads. Each group compares waiting strategy (`Busy`, `Pause`) with different placement strategies (Die, Socket, SMT).

the effect is likely to be more pronounced in thermally constrained systems, where localized thermal hotspots more readily limit boost frequency due to less aggressive cooling solutions or sustained high ambient temperatures [25].

L2: Placement decides who shares the budget. The compute budget is distributed hierarchically across cores, chiplets, and sockets. Service placement therefore determines which applications compete for power, frequency, and thermal headroom, not just cache locality.

3.5 Waiting Trades Latency for Budget

Figure 7 and Figure 8 show the trade-offs between polling strategies across burst scales (μs, ms, s) and busy:idle ratios (1:9, 5:5, 9:1). We dedicate 96 threads to matrix multiplication, while the remaining threads run a polling application that processes headers and computes a pseudo checksum during busy periods, similar to network packet processing.

We report the aggregate throughput of both applications for each waiting strategy. With `Block`, polling threads run with real-time priority and share logical cores with additional matrix multiplication threads. Sleeping polling threads yield to matrix multiplication, and waking polling threads immediately preempt them. Figure 7 normalises throughput to a matrix multiplication running alone on all 192 threads.

Waiting strategy impacts application performance. Matrix multiplication benefits from blocking polling during idle periods. At the second timescale, blocking exhibits only a 5.7% slowdown of baseline throughput when the polling

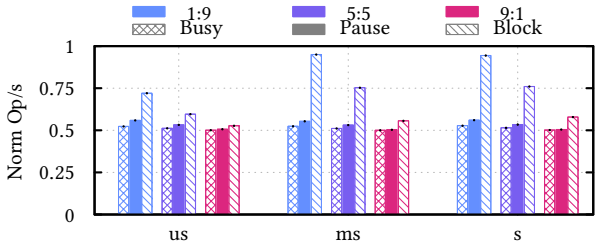

Figure 7: Normalised throughput of scalar matrix multiplication, co-located with bursty polling workloads using different waiting strategies (`Busy`, `Pause`, and `Block`). Results are grouped by burst timescale (μs, ms, and s) and busy:idle ratios (1:9, 5:5, and 9:1). Blocking frees the most application capacity when idle periods are long enough to amortise scheduler overhead, but `Pause` helps application capacity at short timescales.

application is busy for one second and idle for nine. At the microsecond timescale, the overheads are not amortised and blocking incurs a 28.1% slowdown relative to the baseline. In comparison, we measure 47.7% slowdown for busy polling at the microsecond timescale. `Pause` polling does slightly better, at 44.2% slowdown.

Wakeup overheads impact polling performance. The overheads and wakeup latencies of the different strategies influence polling performance. At the microsecond timescale, blocking processes 0.9 million pseudo-packets per second for a 1:9 busy:idle ratio. Polling with `Pause`, in contrast, processes 1.46 million pseudo-packets per second, a 62.2% speedup. At the second timescale, scheduler overheads are amortised and blocking achieves performance comparable to busy and `Pause` polling.

L3: Waiting policy should be adaptive. Waiting spans a spectrum between busy polling and blocking. Different points trade wakeup latency for returned compute budget. Network stacks should adapt their waiting mechanism to expected idle duration instead of treating polling and blocking as fixed alternatives.

L4: Stop paying the cost of complexity at small timescales to eliminate idle cycles. Core reallocation is worthwhile only after scheduler overheads, migration costs, and cache disruption can be amortised. At microsecond timescales, retaining ownership and selecting a better waiting policy to redistribute the fixed budget is often more effective than moving work between cores.

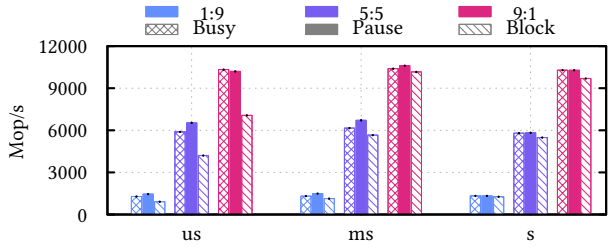

Figure 8: Throughput of a bursty polling application using different waiting strategies (`Busy`, `Pause`, and `Block`), co-located with matrix multiplication. Results are grouped by burst timescale (μs, ms, and s) and busy:idle ratios (1:9, 5:5, and 9:1). Blocking loses polling throughput at the microsecond timescale but becomes competitive at longer timescales. `Pause` achieves performance comparable to `Busy` at small timescales.

4 Outlook and Discussion

Our results suggest that host network stacks need to take the processor's budget into account to increase efficiency and performance.

Toward budget-aware network stacks. Future service-core designs should account for networking in terms of CPU power and thermal budget rather than solely core count. Instead of deciding when to lend a network core to the application, they should decide how much budget networking should consume while meeting latency objectives. Efficient core reallocation [10, 17, 32] is a valuable technique here, especially for managing longer-term workload shifts or locality changes. Waiting policy, placement, and occasional reallocation then balance application throughput against network responsiveness.

The interface gap. Current software offers two extremes: busy polling that retains the core, or blocking that relinquishes it to the scheduler. Modern processors provide intermediate hardware mechanisms that return increasing amounts of compute budget while preserving ownership of the core, but these are difficult to use from userspace and are not exposed through portable interfaces. Service-core network stacks would benefit from a standard "halt-in-place" primitive that separates waiting from core ownership.

Hardware trends. The fixed-budget regime is likely to become more common. Core counts continue to increase while package power grows much more slowly, making power and thermal headroom an increasingly shared resource. Chiplet-based processors further make this budget hierarchical, so placing network services becomes a locality decision and a budget allocation decision.

However, as thermal and power management become increasingly complex and demand faster response, the boundary between hardware and software-managed control must be renegotiated. Prior generation CPUs offered substantial direct software control. For example, the OS can set the frequency of individual cores. Modern SKUs instead place more control in hardware, leaving software to provide advisory hints at best. To illustrate, we initially sought to evaluate the effect of per-core frequency control on the budget model using an AMD EPYC 9655. However, despite exhaustive experimentation, none of the available indirect interfaces allowed software to explicitly clock selected busy cores lower than others. On five-year-old AMD and Intel systems in our cluster, this was possible by changing a single runtime parameter.

Limitations. Our conclusions apply to processors operating near their package power and thermal limits. Outside this regime, the classical model of independent cores remains appropriate. Finally, although the underlying mechanisms are common to modern processors, our evaluation is primarily based on a recent AMD platform. Evaluating other processor families is an important direction for future work.

5 Related Work

Waiting while retaining ownership. Prior work analysed spinning versus blocking for locks [5, 18], and the energy-throughput trade-offs among spinning, `PAUSE`, `MWAIT`, and blocking [8]. Others reduce blocking and wakeup costs [14], exploit `MWAIT` for idle cores [4], or examine how virtualization hides hardware idleness [37]. These reduce the costs of busy polling; we go further, arguing that under a power cap efficient waiting is a transfer rather than a saving, so busy polling is not as wasteful as assumed.

Trading frequency for latency and energy. A large body of work adapts frequency to workload needs, from sub-request DVFS for latency-critical services [13, 19] to application-controlled scaling [6, 12, 30, 36], and feedback-driven C-state control [39]. These treat frequency as a dial on a single core's energy per request. Unlike race-to-idle designs [29] aimed at energy proportionality [3], our regime is limited by a sustained power rate, not an energy quota: lowering a service core's frequency does not merely save energy but frees headroom that hardware redistributes as frequency to application cores. Frequency scaling control provides a natural knob in our proposed model: it controls the sustained rate at which a core spends its budget. We frame waiting depth and execution rate as two controls over one quatity, the budget a service core consumes.

Reclaiming and harvesting idle capacity. Microsecond schedulers reclaim idle service cores by reallocation [10, 17, 32], supported by low-overhead handoffs [2, 16, 24] and

policies for microsecond tasks [28]. We qualify their shared premise: at the power cap a reclaimed core runs at reduced frequency, so recovered work is worth far less than its core count implies, while the idle core it replaced already returned most of its budget for free. Reallocation and harvesting are still worthwhile once idle periods amortise their overhead or below the power cap, but not as the default in the microsecond, power-limited, regime service cores occupy.

6 Conclusion

Modern processors increasingly behave as a shared compute budget rather than a collection of independent cores. In this regime, idle service cores are not necessarily wasted, and reclaiming them at microsecond timescales can add complexity without recovering meaningful compute. We hope this perspective encourages the community to build budget aware network stacks that better exploit large multicore systems.

References

[1] Advanced Micro Devices, Inc. AMD EPYC 9655 processor. https://www.amd.com/en/products/processors/server/epyc/9005-series/amd-epyc-9655.html.

[2] Berk Aydogmus, Linsong Guo, Danial Zuberi, Tal Garfinkel, Dean Tullsen, Amy Ousterhout, and Kazem Taram. Extended user interrupts (xUI): Fast and flexible notification without polling. In *30th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2025.

[3] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12), December 2007.

[4] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *22nd ACM Symposium on Operating Systems Principles*, SOSP, 2009.

[5] Leonid B. Boguslavsky, Karim Harzallah, Alexander Y. Kreinin, Kenneth C. Sevcik, and Alexander Vainshtein. Optimal strategies for spinning and blocking. *Journal of Parallel and Distributed Computing*, 1994.

[6] DPDK Authors. Data plane development kit documentation. 58. power management. https://doc.dpdk.org/guides-24.07/prog_guide/power_man.html.

[7] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *38th Annual International Symposium on Computer Architecture*, ISCA, 2011.

[8] Babak Falsafi, Rachid Guerraoui, Javier Picorel, and Vasileios Trigonakis. Unlocking energy. In *2016 USENIX Annual Technical Conference*, ATC, 2016.

[9] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz André Barroso. Power provisioning for a warehouse-sized computer. In *34th Annual International Symposium on Computer Architecture*, ISCA, pages 13–23, 2007.

[10] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2020.

[11] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4):6–15, July 2011.

[12] Tomas Hruby, Herbert Bos, and Andrew S. Tanenbaum. When slower is faster: On heterogeneous multicores for reliable systems. In *2013 USENIX Annual Technical Conference*, ATC, 2013.

[13] Chang-Hong Hsu, Yunqi Zhang, Michael A. Laurenzano, David Meisner, Thomas Wenisch, Ronald G. Dreslinski, Jason Mars, and Lingjia Tang. Reining in long tails in warehouse-scale computers with quick voltage boosting using adrenaline. *ACM Transactions on Computer Systems*, 35(1), March 2017.

[14] Jack Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. A case against (most) context switches. In *18th Workshop on Hot Topics in Operating Systems*, HOTOS, 2021.

[15] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, July 2024.

[16] Rishabh Iyer, Musa Unal, Marios Kogias, and George Candea. Achieving microsecond-scale tail latency efficiently with approximate optimal scheduling. In *29th ACM Symposium on Operating Systems Principles*, SOSP, 2023.

[17] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for microsecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2019.

[18] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan S. Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *13th ACM Symposium on Operating Systems Principles*, SOSP, 1991.

[19] Harshad Kasture, Davide B. Bartolini, Nathan Beckmann, and Daniel Sanchez. Rubik: fast analytical power management for latency-critical systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, 2015.

[20] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP acceleration as an OS service. In *14th ACM European Conference on Computer Systems*, EuroSys, 2019.

[21] Alok Gautam Kumbhare, Reza Azimi, Ioannis Manousakis, Anand Bonde, Felipe Frujeri, Nithish Mahalingam, Pulkit A. Misra, Seyyed Ahmad Javadi, Bianca Schroeder, Marcus Fontoura, and Ricardo Bianchini. Prediction-Based power oversubscription in cloud platforms. In *2021 USENIX Annual Technical Conference*, ATC, pages 473–487, 2021.

[22] Malte-Christian Kuns, Hannes Tröpgen, and Robert Schöne. An analysis of user-space idle state instructions on x86 processors. In *Proceedings of the 16th ACM/SPEC International Conference on Performance Engineering*, ICPE '25, 2025.

[23] Shaohong Li, Xi Wang, Xiao Zhang, Vasileios Kontorinis, Sreekumar Kodakara, David Lo, and Parthasarathy Ranganathan. Thunderbolt: Throughput-Optimized, Quality-of-Service-Aware power capping at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, pages 1241–1255, 2020.

[24] Jiazhen Lin, Youmin Chen, Shiwei Gao, and Youyou Lu. Fast core scheduling with userspace process abstraction. In *30th ACM Symposium on Operating Systems Principles*, SOSP, 2024.

[25] Rui Lu and Dan Wang. A thermal-aware workload scheduler for high-performance LLM inference in cooling-regulated datacenters. In *HotCarbon '25: Workshop on Sustainable Computer Systems*, HotCarbon, 2025.

[26] Yenai Ma, Leila Delshadtehrani, Cansu Demirkiran, José L. Abellán, and Ajay Joshi. TAP-2.5D: A thermally-aware chiplet placement methodology for 2.5D systems. In *Proceedings of the 2021 Design, Automation & Test in Europe Conference & Exhibition*, DATE, pages 1246–1251, 2021.

[27] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In *27th ACM Symposium on Operating Systems Principles*, SOSP, 2019.

[28] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. Efficient scheduling policies for Microsecond-Scale tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, pages 1–18, 2022.

[29] David Meisner, Brian T. Gold, and Thomas F. Wenisch. Powernap: eliminating server idle power. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, 2009.

[30] Louis-Marie Nicolas, Salim Mimouni, Philippe Couvée, and Jalil Boukhobza. Characterizing the use of DVFS for HPC I/O optimization: A microbenchmarking approach. In *5th Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*, CHEOPS, 2025.

[31] Zhixiong Niu, Hong Xu, Peng Cheng, Qiang Su, Yongqiang Xiong, Tao Wang, Dongsu Han, and Keith Winstein. NetKernel: Making network stack part of the virtualized infrastructure. In *2020 USENIX Annual Technical Conference*, ATC, 2020.

[32] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, pages 361–378, 2019.

[33] Giuseppe Romano, Aakrati Jain, Nima Dehmamy, Cheng Chi, and Xin Zhang. DiffChip: Thermally aware chip placement with automatic differentiation. In *Proceedings of the 75th IEEE Electronic Components and Technology Conference*, ECTC, pages 221–226, 2025.

[34] Matheus Stolet, Liam Arzola, Simon Peter, and Antoine Kaufmann. Tail contagion: Sub-microsecond time protection in shared software network datapaths. *arXiv preprint arXiv:2309.14016*, 2026. https://arxiv.org/abs/2309.14016.

[35] Matheus Stolet, Simon Peter, and Antoine Kaufmann. Chamelio: A fast shared cloud network stack for isolated tenant-defined protocols, 2026. https://arxiv.org/abs/2604.22603.

[36] Jons-Tobias Wamhoff, Stephan Diestelhorst, Christof Fetzer, Patrick Marlier, Pascal Felber, and Dave Dice. The TURBO diaries: Application-controlled frequency scaling explained. In *2014 USENIX Annual Technical Conference*, ATC, 2014.

[37] Yun Wang, Xingguo Jia, Ben Luo, Kenan Liu, Shengdong Dai, Jingdong Han, Weihao Chen, Yicheng Gu, Xingzi Yu, Yibin Shen, Jiesheng Wu, Zhengwei Qi, and Haibing Guan. What are you (M)waiting for: The hidden cost of idle in the hyperscale cloud. In *20th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2026.

[38] Qiang Wu, Qingyuan Deng, Lakshmi Ganesh, Chang-Hong Hsu, Yun Jin, Sanjeev Kumar, Bin Li, Justin Meza, and Yee Jiun Song. Dynamo: Facebook's data center-wide power management system. In *43rd Annual International Symposium on Computer Architecture*, ISCA, pages 469–480, 2016.

[39] Xin Zhan, Reza Azimi, Svilen Kanev, David Brooks, and Sherief Reda. CARB: A c-state power management arbiter for latency-critical workloads. *IEEE Computer Architecture Letters*, 16(1):6–9, 2017.