DOI:10.1145/3232559

Queueing theoretic models can guide design trade-offs in systems targeting tail latency, not just average performance.

BY CHRISTINA DELIMITROU AND CHRISTOS KOZYRAKIS

Amdahl's Law for Tail Latency

TRANSLATING THE IMPACT of Amdahl's Law on tail latency provides new insights on what future generations of data-center hardware and software architectures should look like. The emphasis on latency, instead of just throughput, puts increased pressure on system designs that improve both parallelism and single-thread performance.

Computer architecture is at an inflection point. The emergence of warehouse-scale computers has brought large online services to the forefront in the form of Web search, social networks, software-as-a-service, and more. These applications service millions of user queries daily, run distributed over thousands of machines, and are concerned with tail latency (such as the 99th percentile) of user requests in addition to high throughput.[6] These characteristics represent a significant departure from previous systems, where the performance metric of interest was only throughput, or, at most, average latency. Optimizing for tail latency is already changing the way we build operating systems, cluster managers, and data services.[7,8] This article investigates how the focus on tail latency affects hardware designs, including what types of processor cores to build, how much chip area to invest in caching structures, how much resource interference between services matters, how to schedule different user requests in multicore chips, and how these decisions interact with the desire to minimize energy consumption at the chip or data-center level.[2]

While the precise answers will come from detailed experiments with both simulated and real systems, there is great value in having an analytical framework that identifies the major trade-offs and challenges in latency-sensitive cloud systems. We aim here to complement the previous analyses on Amdahl's Law for parallel and multicore systems[1,11] by designing a model that draws from basic queueing theory

» key insights

- Optimizing for tail latency makes Amdahl's Law more consequential than when optimizing for average performance.
- Queueing theory can provide accurate first-order insights into how hardware for future interactive services should be designed.
- As service responsiveness and predictability become more critical, finding a balance between compute and memory resources likewise becomes more critical.

Analytical Framework

Amdahl's Law describes the speedup of a program when a fraction f of the computation is accelerated by a factor S. Speedup is then defined as

$$Speedup(f,S) = \frac{1}{(1-f)+\frac{f}{S}}$$

In a multi-core machine, Amdahl's Law captures the benefit from multiple cores in average performance. While this interpretation is still relevant, it is, by itself, insufficient for describing tail latency requirements. To bridge the gap we build upon ideas from queueing theory, which provides a framework to reason about task-arrival rates, service times, and end-to-end response times. Simple models (such as M/M/1 and M/M/k) are particularly attractive for first-order performance calculations because they can concisely describe performance in closed-form expressions.

M/M/1 model. We start with one of the simplest queueing models: the M/M/1 queue, modeling a system in which a single server processes incoming tasks. Tasks arrive under a Poisson process with rate λ. The service times also follow an exponential distribution, with rate parameter μ and mean service time $T_s = 1/\mu$ (μ=*perf*(*r*) in the main text of the article. A larger μ means a more powerful server and results in lower latency. Tasks are processed in a simple first-in-first-out order. This simple queueing system is stable when $\mu > \lambda$. In contrast, when $\mu > \lambda$, queued tasks keep increasing, leading to instability. The load of the system is defined as $\rho = \lambda/\mu$. Given these definitions, the mean number of tasks in the system is

$$E[N] = \frac{\rho}{1-\rho}$$

where N is a random variable for the number of tasks. Likewise, the mean of task response time (using random variable R) is

$$E[R] = E[N]\lambda^{-1} = \frac{1}{\mu - \lambda}$$

and the ρ-th percentile of response time is

$$m_p = -\frac{ln(1-p/100)}{\mu - \lambda}$$

Figure 1a outlines the 99th percentile of request latency as a function of the service rate μ. As μ increases, tail latency drops both at low and high load.

M/M/k model. We now extend the M/M/1 model to a more realistic system with k equivalent servers in order to model a multicore machine. Tasks are now added to a single, shared queue, where servers draw them from for processing. As with the M/M/1 model, tasks arrive under a Poisson process with arrival rate λ and each server processes tasks with service rate μ. Closed-form solutions for the mean response time and response-time percentiles exist but are more complicated than in the M/M/1 model. Specifically, system load is $\rho = \lambda/(k\mu)$. The probability that a new task must be enqueued is given by Erlang's C formula

$$C(k,\lambda/\mu) = \frac{\left(\frac{(k\rho)^k}{k!}\right)\left(\frac{1}{1-\rho}\right)}{\sum_{c=0}^{k-1}\frac{(k\rho)^c}{c!} + \left(\frac{(k\rho)^k}{k!}\right)\left(\frac{1}{1-\rho}\right)}$$

and the mean number of tasks in the system

$$E[N] = \frac{\rho}{1-\rho}C(k,\lambda/\mu) + k\rho$$

The average response time is

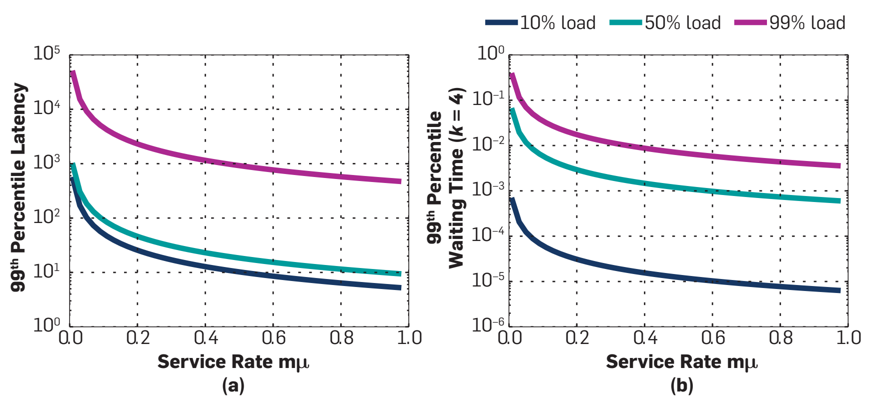
$$E[R] = \frac{C(k,\lambda/\mu)}{k\mu - \lambda} + \frac{1}{\mu}$$

Finally, the p-th percentile of queueing time is

$$w_p = -\frac{ln(1-p/(100C(k,\lambda/\mu)))}{k\mu - \lambda}$$

Figure 1b outlines how the 99th percentile of queueing time correlates to the service rate μ for one and four servers. Higher service rates correspond to less time spent by requests in the queue. We use the M/M/k model for analysis of system trade-offs unless otherwise specified. In the article's section on validation, we verify that this model closely reflects real system behavior. For applications with non-Poisson arrival- and service-time distributions, more general queueing models may be needed (such as the G/G/k model).[10,24] For more complex applications (such as multi-tier services), system architects would need a more sophisticated analytical model (such as a queueing network).

Figure 1. Building system insights from queueing theory: (a) 99th percentile response time in an M/M/1 model; and (b) 99th percentile queueing time in an M/M/4 model as a function of μ.



(see Figure 1 in the sidebar "Analytical Framework") and can provide first-order insights on how design decisions interact with tail latency. As was the case with the previous analyses based on Amdahl's Law, our model has significant implications for processor designs for cloud servers.

While analytical models help draw first-order insights, they run the risk of not accurately reflecting the complex operation of a real system. In Figure 2, we show a brief validation study of the queueing model, as discussed in the sidebar, with {1, 4, 8, 16} compute cores against a real instantiation of memcached, a popular in-memory, key-value store, with the same number of cores. We set the mean interarrival rate and service time of the queueing model based on the measured times with memcached. In both cases, when providing memcached with exponentially distributed input load, the memcached request latency is close to the one estimated by the queueing model across load levels.

Cost Model

Since hardware resources are not infinite, this analysis requires a cost model that relates resource usage to performance. We use a model similar to the one used by Hill and Marty[11] to extend Amdahl's Law to multicore chips. That is, we assume a given multicore chip is limited to R base core equivalents (BCE) units. This limitation represents area or power-consumption constraints in the chip design. The BCE is an abstract cost unit that captures processor resources and caches but does not share resources (such as interconnection networks and memory controllers). As in Hill and Marty,[11] we assume these resources are fairly constant in the system variations we examine. A baseline core that consumes 1BCE unit achieves performance of $perf(1)=1$. Chip architects can build more powerful cores by dedicating $r \in [1,R]$ resource units to each core to achieve performance $perf(r)$, where $perf(r)$ is the rate parameter μ in our performance model. Larger cores have higher service rate μ, which is inversely related to tail latency, as discussed in the sidebar. If performance increases superlinearly with resources, then more cores are always better. In practice $perf(r) < r$, creating trade-offs between opting for few brawny or many wimpy cores. By default, we follow Shekhar Borkar[3] and use $perf(r) = sqrt(r)$ but have also investigated how higher roots affect the corresponding insights.

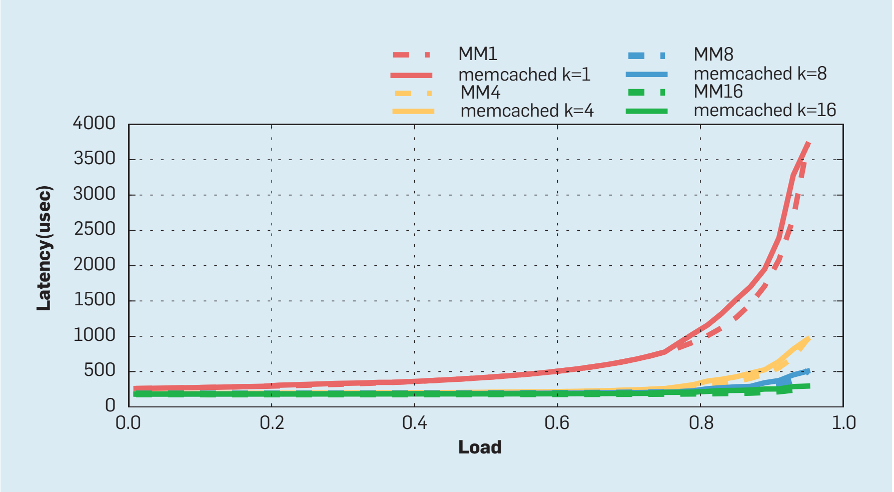
Brawny Versus Wimpy Cores

We first examine a system where all cores are homogeneous and have identical cost. An important question the designer must answer is: Given a constrained aggregate power or area budget, should architects build a few large cores or many small cores? The answer has been heavily debated in recent years in both academia and industry,[4,12,14,17,19,22] as it relates to the introduction of new designs (such as the ARM server chips and throughput processors like Xeon Phi).

Assuming the total budget is R = 100BCEs, an architect can build 100 basic cores of 1BCE each, 25 cores of 4BCEs each, one large core of 100BCEs, or in general R/U cores of U units each, as shown in Figure 3. We consider an online service workload with tail latency quality-of-service (QoS) constraints. QoS is defined as a function of the mean service time T_s of the 100BCE machine. For example, a very strict QoS target would require the 99th percentile of request latency to be T_s. This means the time between arrival and completion of 99% of requests must be less or equal to the machine's mean service time, allowing no tolerance for queueing or service-time variability. More relaxed QoS targets are defined as multiples of T_s: $QoS = \alpha T_s$, $\alpha \in [5, 10, 50, 100]$. Figure 4a shows how throughput in queries per second (QPS) changes for different latency QoS targets, under the M/M/N queueing model described in the sidebar. Throughput of 100QPS for QoS=10Ts means the system achieved 100QPS for which the 99th latency percentile is $10T_s$. The x-axis captures the size of selected cores, moving from many small cores on the left side to a single core of 100BCEs on the right side. We examine all core sizes from 1BCE up to 100BCEs in increments of a single resource unit. In configurations with multiple cores, throughput is aggregated across all cores. The discontinuities in the graph are an artifact of the limited resource budget and homogeneous design; for example, for U = 51, an architect can build a single 51BCE core, while 49 resource units remain unused. Throughput for $10T_s$ for cores greater than 7BCE overlaps with 100Ts, as does throughput for $5T_s$ for cores of more than 12BCEs.

Finding 1. Very strict QoS targets put a lot of pressure on single-thread performance. When QoS = T_s or 5 T_s, cores smaller than 22BCEs or 12BCEs, respectively, achieve zero QPS for which the tail latency satisfies the QoS target. This happens because the cores are too weak to handle variability in service time even in the absence of queueing, and the queueing naturally occurs when cores operate close to saturation. This result means that, for services with extremely low-latency requirements (such as in-memory caching and in-memory distributed storage),[21] architects must focus on improving

Figure 2. Validation of the queueing model against a real instantiation of an in-memory key-value store (memcached) for {1,4,8,16} cores.



single-thread performance even at high cost. At the same time, some core parallelism is needed. A single 100BCE core performs significantly worse than four 25BCE cores. This finding is in agreement with industry concerns about the performance of small cores with warehouse-scale services.[12] The need for high single-thread performance also motivates application- or domain-specific accelerators as a more economical way of improving performance than incremental out-of-order core optimizations.

Finding 2. At lower latency constraints, architects should look for ways to balance optimizations for single-thread performance and request-level parallelism. At lower QoS targets, a larger set of medium-size cores achieves the best performance. For example, 7BCE cores are optimal for QoS = $10T_s$. For applications with moderate latency requirements (such as Web search and Web servers), architects should seek to balance improvements in single-thread performance (instruction-level parallelism) and multi-core performance (request-level parallelism). Increasing single-thread performance at high cost yields diminishing returns in this case. Nevertheless, a large pool of wimpy cores—1BCE—is optimal only when applications have no latency constraints, as with long data mining queries or log-processing requests. With QoS = $100T_s$, applications are essentially throughput-limited and perform best with many wimpy cores.

These findings highlight a disparity between optimal system design when optimizing for throughput versus when optimizing for tail latency. For example, in a homogeneous system where throughput is the only performance metric of interest and parallelism is plentiful, the smallest cores achieve the best performance; see the 1BCE cores in Figure 4a. In comparison, when optimizing for throughput under a tail latency constraint, the optimal design point shifts toward larger cores, unless the latency constraint relaxes significantly.

Finding 3. Limited parallelism also calls for more powerful cores. So far we have assumed all user requests are independent and perfectly parallelizable, though it is rarely the case in practice. Requests are often dependent on each other and on system issues like connection ordering and locks for writes causing serialization. The growing trend of breaking complex services down to smaller components (microservices) will only make the problem of request dependencies more common. This brings up the caveat of Amdahl's Law. To what extent are the previous findings accurate when parallelism is limited? Figure 4b shows the case of a reasonable QoS ($10T_s$) with $f \in \{50\%, 90\%, 99\%, 100\%\}$. When, for example, the parallel fraction of the computation f is 90%, 10% of requests are serialized. As a result, while optimal performance was previously achieved with seven BCE cores, the optimal core size now shifts to 25 BCEs. Limited parallelism also affects throughput-centric systems,[11] with more powerful cores outperforming wimpy cores in applications with serial regions. Using Hill's and Marty's model[11] with a 100BCE budget and 10% serialization, an architect would determine that 10BCE cores are optimal for throughput, a less aggressive increase in core size than when optimizing for latency. As parallelism decreases further, more performant cores are needed to drive down tail latency. When 50% of execution is serial, a single 100BCE core is optimal,

Figure 3. Homogeneous server configurations for a budget of $R = 100$ resource units: (a) 100 1BCE cores; (b) 25 4BCE cores; and (c) one 100BCE core.

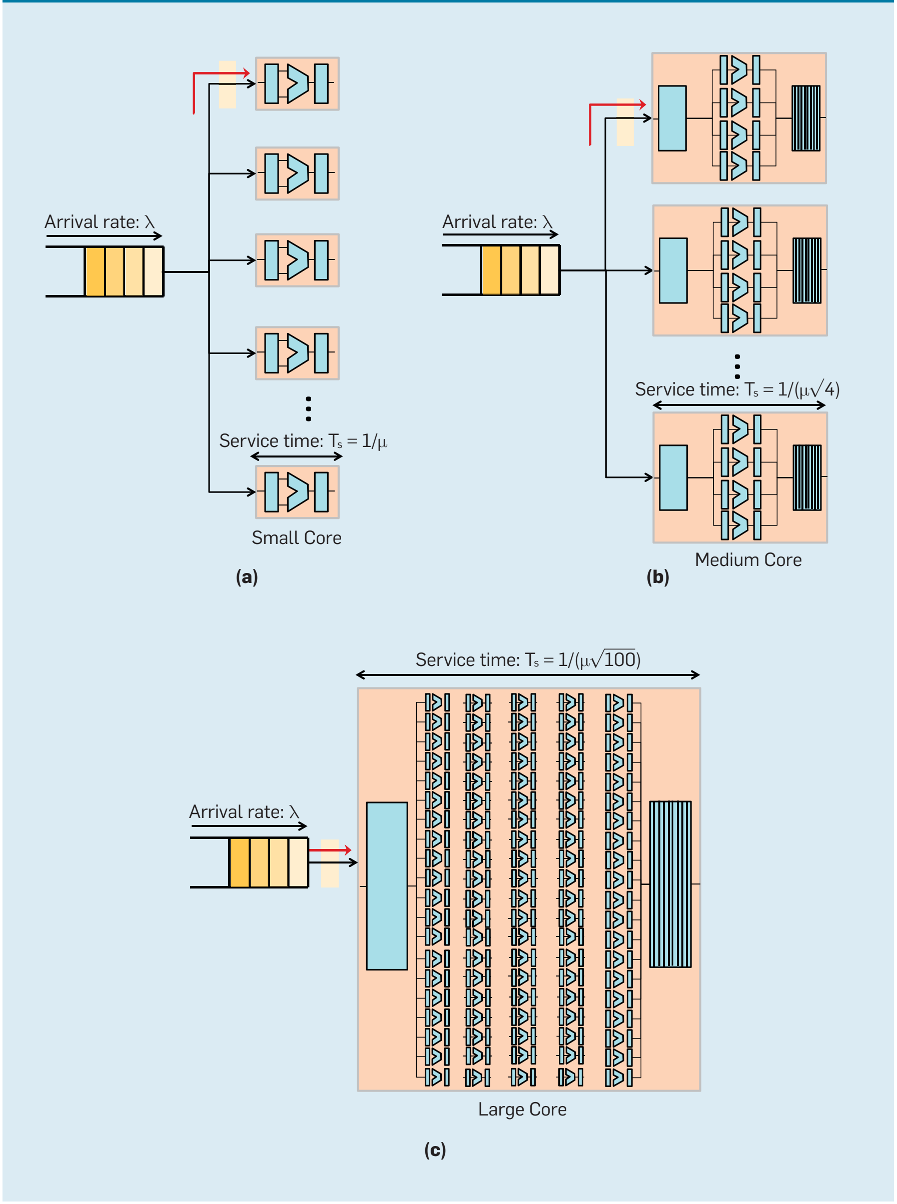
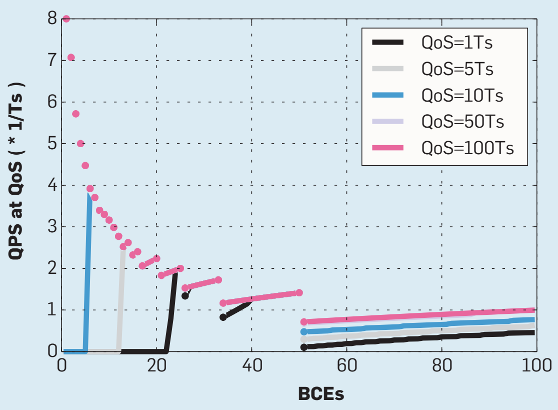
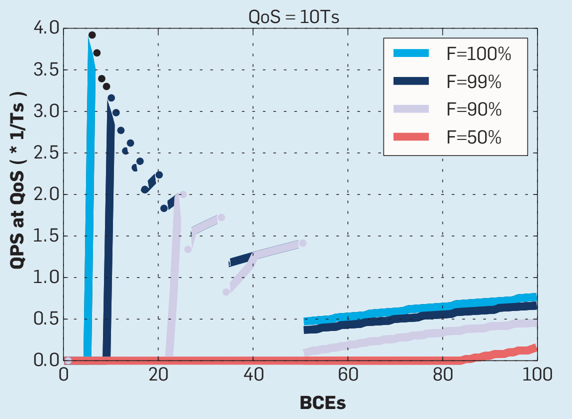

Arrival rate: λ
Service time: $T_s = 1/\mu$
Small Core
(a)
Arrival rate: λ
Service time: $T_s = 1/(\mu\sqrt{4})$
Medium Core
(b)
Service time: $T_s = 1/(\mu\sqrt{100})$
Arrival rate: λ
Large Core
(c)

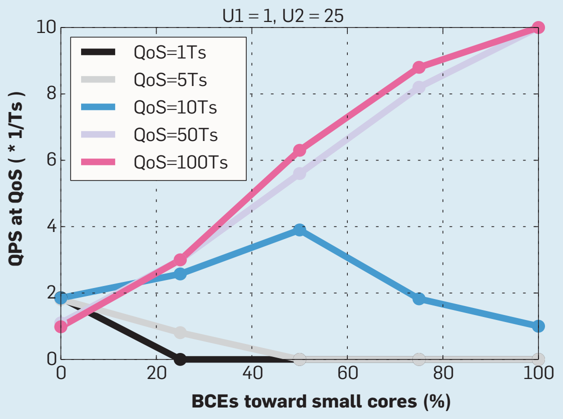
Figure 4. Studies on big versus small cores, core heterogeneity, and caching using the queueing model.



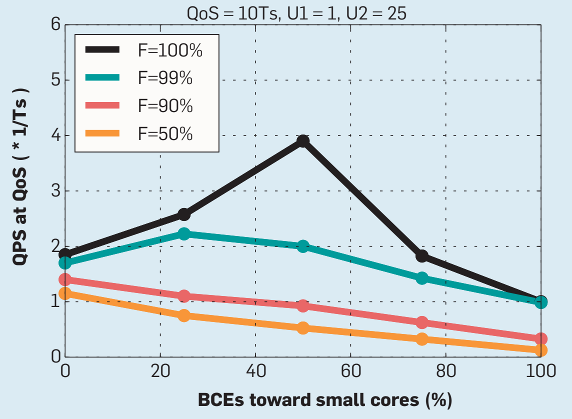
(a) Throughput (QPS) under a tail latency constraint as a system architect increases the resources per core when parallelism is unlimited;



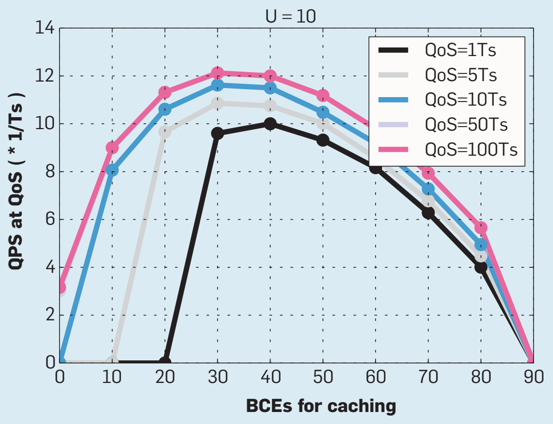
(b) Throughput under a tail latency constraint when parallelism is not plentiful;



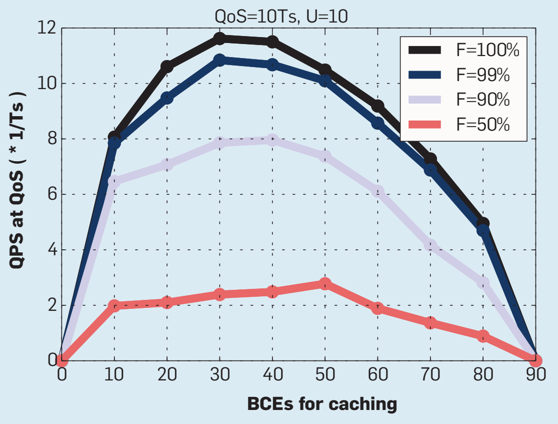
(c) Throughput (QPS) under a tail latency constraint as a system architect increases the resources for small cores (U1=1) under the assumption of unlimited parallelism;



(d) Throughput under a tail latency constraint when parallelism is limited;



(e) Throughput (QPS) under a tail latency constraint as system architect increases resources for caching, as opposed to compute when parallelism is unlimited;



(f) Throughput under a tail latency constraint when parallelism is not plentiful.

a dramatic shift from the unlimited-parallelism case; overall throughput is also an order of magnitude lower. Quantifying the degree of parallelism in latency-critical services is essential when deciding how to build the underlying hardware. At the same time, computer scientists should strive to remove serialization across the system stack—at the application level by developing tracing and monitoring systems that detect and minimize cross-service dependencies, at the operating system by minimizing the need for lock serialization, and at the architecture level by investing in methods that increase single-thread performance and intra-request parallelism.[9]

These findings remain consistent for *perf* (r) scaling with the square, cubic,

Figure 5. Heterogeneous server configuration with 25BCE large cores and 1BCE small cores.

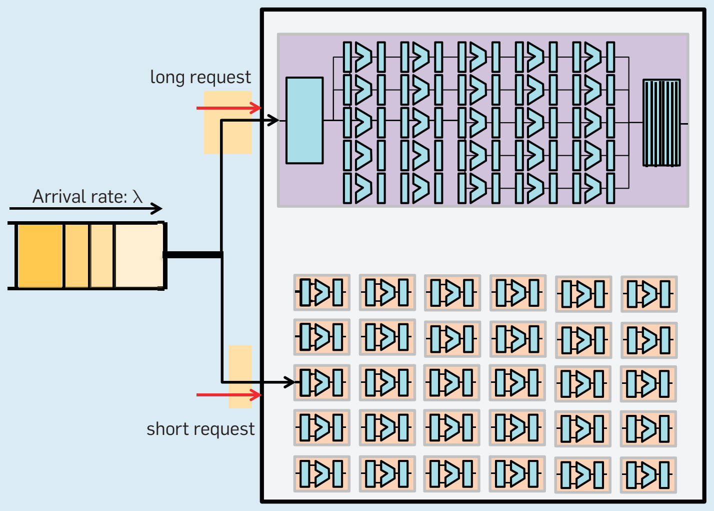
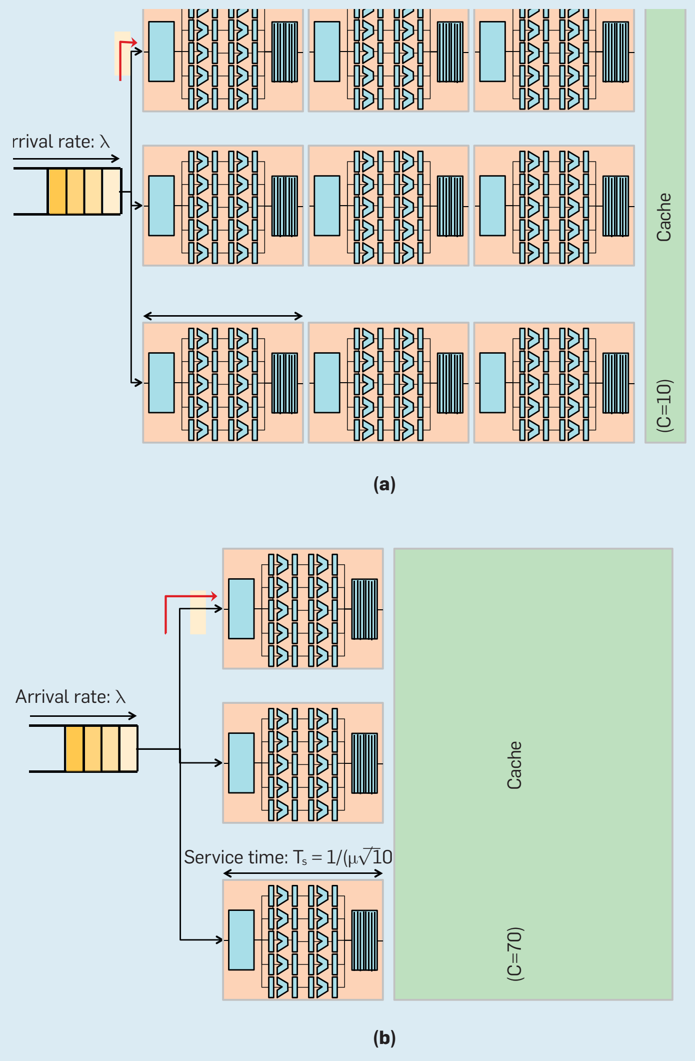


Figure 6. Server configurations with 10BCE cores when dedicating (a) 10 resource units and (b) 70 resource units toward caching.



and fourth root of r. Beyond that point, optimal design favors smaller cores.

Core Heterogeneity

The previous section explored the trade-offs between powerful, brawny cores and power-efficient, wimpy cores. Neither type of core provides high efficiency across a wide range of QoS targets, raising several obvious questions, including: Should an architect combine multiple core types in the same system, as is already the norm in multicore chips for mobile systems? How should architects determine the size of these cores? And at what ratio should they use them? Determining the right mix of large-versus-little cores, as well as devising schedulers that take advantage of heterogeneous cores, especially in the presence of heterogeneous load, has been a notably active topic of research in computer architecture in recent years.[5,9,15] Figure 4c shows the QPS under various QoS targets for a set of heterogeneous designs. In all cases, the system has two core configurations: small cores with $U = 1$, benefiting applications with relaxed QoS, and big cores with $U = 25$, benefiting applications with strict QoS. The system also receives two exponentially distributed input request streams, one with short and the other with long mean-service-time requests, and design a simple heterogeneity-aware scheduler that routes long requests to big cores and short requests to small cores. Requests are admitted to a single queue, as in Figure 5, and the ratio of long-to-short requests is, for now, 1:1. Figure 5 starts with all big cores at the leftmost point of the x-axis, explores the heterogeneous space, and ends with all small cores at the rightmost point.

Finding 4. Figure 4c captures a surprising trend. For strict QoS targets, like $1 \cdot T_s$, homogeneous systems with all big cores achieve optimal performance. In contrast, for very relaxed QoS targets, like $100T_s$, using all small cores achieves the best performance. However, for QoS targets in the middle (such as $10T_s$), heterogeneous systems, coupled with heterogeneity-aware schedulers, outperform their homogeneous counterparts. This result is especially true when the ratio of big to small cores matches the ratio of long-to-short requests. Varying the request ratio affects these findings significantly. The further away the ratio of long-to-short requests is from the ratio of big-to-small cores the more homogeneous systems outperform their heterogeneous counterparts. This result means that for heterogeneous architectures to make sense the system must closely track the input load and adjust to its changes, a common phenomenon in large-scale online services.[18]

Finding 5. We have again assumed unlimited request parallelism. Once serialization between requests is introduced, the optimal operation point shifts. Figure 4d shows QPS under various tail-latency QoS targets for increasing values of $f \in \{50\%, 90\%, 99\%, 100\%\}$. Where previously homogeneity outperformed heterogeneous designs for extreme QoS requirements—very strict and very relaxed—now takes the lead heterogeneity. For example, for a moderate QoS target of $10T_s$ and $f = 0.9$ a single big core achieves optimal performance, compared to the 50:50 mix in Figure 4c. In general, the more parallelism is limited the more the optimal operation point shifts left, with more big and fewer smaller cores. This is in agreement with Hill's and Marty's observations,[11] with the added implication that latency considerations cause a more rapid shift toward larger cores than when throughput is the only performance metric of interest. For example, when $f = 0.9$ and the system optimizes only for throughput, two 50BCE cores achieve the best performance under Hill's and Marty's model. As before, this result highlights the importance of quantifying the degree of parallelism in interactive applications. It also establishes that, even with limited parallelism, scheduling that takes into account the different capabilities of available hardware is essential for harnessing the potential of hardware heterogeneity.

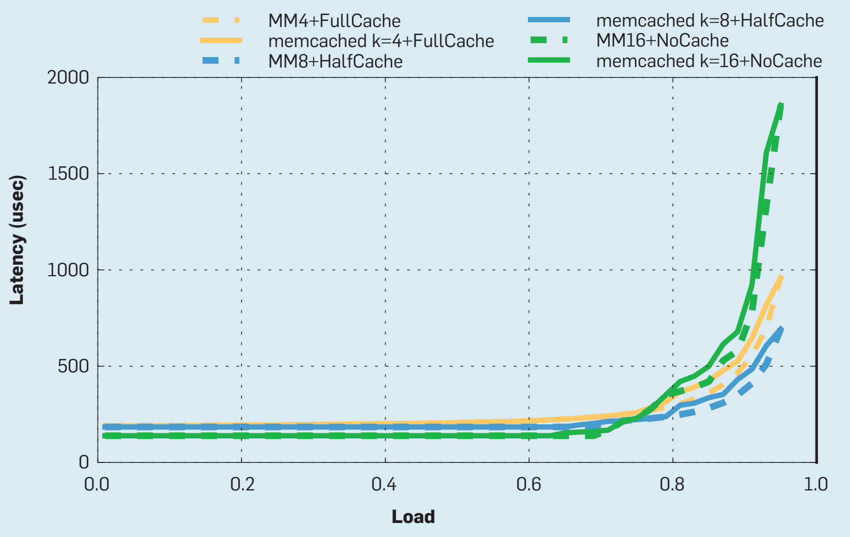
Caching

Architects constantly deal with the trade-off of using the limited resources for compute or caching. Larger caches help avoid the long latencies of main memory but draw significant static power and reduce the amount of resources available for compute cores; see Figure 6 for two characteristic configurations. Using the same total budget as before—$R = 100$—we explore how QPS under a tail-latency constraint changes as a fraction $C \in [0, 90]$ of resources goes toward building caches, as opposed to cores. We use 10BCE cores, benefitting applications with moderately strict QoS targets; Figure 4e shows this trade-off. On the leftmost point of the x-axis all resources are dedicated to building cores. On the rightmost point, 90% of resources go toward building caches and the remaining 10% toward building cores, one 10BCE core in this case. Increasing caching by 10BCE results in one fewer core in the system. We assume caches improve service time under a $sqrt(C)$ function, meaning $T_s0 = T_s = sqrt(C)$.[23] We validate the selection of the scaling factor against a real installation of memcached where the allocated last-level cache partition is adjusted using Intel's Cache Allocation Technology. As the number of used cores increases, the allocated cache capacity decreases. Figure 7 outlines that the difference between the analytical model and the real system is, in general, marginal. The findings reported in Figure 4e remain consistent for scaling functions until the seventh root of C, which corresponds to progressively lower benefits from caching, causing the optimal point to shift increasingly to the left.

Finding 6. For services with strict tail-latency requirements that exhibit locality, the benefit from caching is critical to achieving QoS. For strict QoS constraints (such as QoS = T_s), at least $C = 20$ units are needed to lower the core's service time in a way that achieves QPS under the tail-latency constraint.[16,20] Moderately increasing caching resources beyond $C = 20$ units further improves performance, as larger fractions of the working set fit in the last-level cache;[16] that is, more requests enjoy the shorter processing time of caches for the purpose of the queueing model. However, the benefits diminish beyond $C = 40$, and performance degrades rapidly as compute resources become insufficient.[16] Existing server chips dedicate one-third to one-half of their area budget to caches. Our analysis indicates this trend will continue.

Finding 7. For relaxed QoS targets, caching is less critical. Since smaller cores are sufficient for achieving the QoS constraints in this case, and although caching is still beneficial, moderate cache provisioning (such as $C = 10$ units to 30 units) yields most of its potential performance benefits. Increasing caching units to $C = 40$ has no effect on performance, and further increase degrades performance. Architects should focus instead on exploiting request parallelism in a way that keeps the large number of smaller cores busy.[12,16]

Figure 7. Validation of the queueing model against a real instantiation of an in-memory key-value store (memcached) with increasing caching and reduced compute resources.



Finding 8. Limited parallelism highlights the importance of increased caching. Figure 4f reports the performance for a moderate QoS target of $10T_s$ and increasing values of $f \in [50\%, 90\%, 99\%, 100\%]$. When 10% of the requests need to be serialized, the optimal point for caching is $C = 40$ units compared to $C = 30$ units with unlimited parallelism. Serialized execution requires higher single-thread performance, and larger on-chip caches is one way to achieve such performance.

Discussion

The models we offer here aim to provide first-order insight into how system design decisions affect tail latency and throughput in QoS-constrained services. These models do not capture every aspect of a data-center machine or application.[13] For example, while we can arbitrarily scale service times using the presented queueing model, system call and RPC overheads in real systems have hard lower limits. Likewise, software, especially in cloud applications, is not static. These frequent changes in cloud environments affect the degree of dependencies across requests, in terms of both the request fanout and the dependencies across components of a service (such as in microservices-based cloud applications). A more sophisticated model that captures such dependencies, potentially through a queueing network, can provide more accurate performance estimations at the cost of greater complexity. Finally, in hardware, architects cannot build cores with arbitrarily higher performance by simply adding more resources. They must also account for such factors as locality, coherence, and memory scheduling absent from our current model.

We see queueing theoretic models as a starting point for using queueing theory principles to draw insights into system design. We hope this analysis motivates researchers to develop more sophisticated models that address the limitations we have identified and, more important, the hardware and software that can achieve the performance requirements we highlighted.

Conclusion

Amdahl's Law is as pervasive when it comes to tail latency as it has been for traditional systems. Our goal here has been to offer a simple, intuitive, practical model that can lend first-order insights into which optimizations make sense when an application cares about tail performance. Using it, we have shown the overarching trade-offs in large-versus-small-core systems, heterogeneity, and caching. We encourage computer systems researchers to expand this model to express more sophisticated systems and studies.

Acknowledgments

We thank Mark Hill, Partha Ranganathan, Daniel Sanchez, and the anonymous reviewers for their helpful feedback on earlier drafts of this article.

References

1. Amdahl, G.M. Validity of the single-processor approach to achieving large-scale computing capabilities. In *Proceedings of the Spring Joint Computer Conference* (Atlantic City, NJ, Apr. 18–20). AFIPS ACM Press, New York, 1967, 483–485.
2. Barroso, L. and Hölzle, U. The case for energy-proportional computing. *Computer 40*, 12 (Dec. 2007), 33–37.
3. Borkar, S. Thousand-core chips: A technology perspective. In *Proceedings of the 44th Annual Design Automation Conference* (San Diego, CA, June 4–8). ACM Press, New York, 2007, 746–749.
4. Chen, S., Galon, S., Delimitrou, C., Manne, S., and Martinez, J.F. Workload characterization of interactive cloud services on big and small server platforms. In *Proceedings of the IEEE International Symposium on Workload Characterization* (Seattle, WA, Oct. 1–3). IEEE Press, 2017, 125–134.
5. Craeynest, K., Jaleel, J. et al. Scheduling heterogeneous multi-cores through performance impact estimation (pie). In *Proceedings of the 27th International Conference of the International Society for Computers and Their Applications* (Las Vegas, NV, Mar. 12–14). International Society for Computers and Their Applications, Winona, MN, 2012, 213–224.
6. Dean, J. and Barroso, L.A. The tail at scale. *Commun. ACM 56*, 2 (Feb. 2013), 74–80.
7. Delimitrou, C. and Kozyrakis, C. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, TX, Mar. 16–20). ACM Press, New York, 2013.
8. Delimitrou, C. and Kozyrakis, C. Quasar: Resource-efficient and QoS-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, UT, Mar. 1–5). ACM Press, New York, 2014.
9. Haque, Md. E., Eom, Y.h., He, Y., Elnikety, S., Bianchini, R., and McKinley, K.S. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey, Mar. 14–18). ACM Press, New York, 2015, 161–175.
10. Harchol-Balter, M. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action.* Cambridge University Press, Cambridge, U.K., 2013.
11. Hill, M. and Marty, M.R. Amdahl's Law in the multicore era. *IEEE Computer 41*, 7 (July 2008), 33–38.
12. Hölzle, U. Brawny cores still beat wimpy cores, most of the time. *IEEE Micro 30*, 4 (July-Aug. 2010), 20–24.
13. Kanev, S., Darago, J.P., Hazelwood, K., Ranganathan, P., Moseley, T., Wei, G.-Y., and Brooks, D. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, OR, June 13–17). 2015, 158–169.
14. Khubaib, M., Suleman, A., Hashemi, M., Wilkerson, C., and Patt, Y.N. Morphcore: An energy-efficient microarchitecture for high-performance ILP and high-throughput TLP. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture* (Vancouver, B.C., Dec. 1–5). IEEE Computer Society, Washington, D.C., 2012, 305–316.
15. Li, J., Agrawal, K., Elnikety, S., He, Y., Lee, I-T.A., Lu, C., and McKinley, K.S. Work stealing for interactive services to meet target latency. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Barcelona, Spain, Mar. 12–16). ACM Press, New York, 2016, 1–13.
16. Li, S., Lim, H., Lee, V.W., Ahn, J.H., Kalia, A., Kaminsky, M., Andersen, D.G., Seongil, O., Lee, S., and Dubey, P. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, OR, June 13–17). ACM Press, New York, 2015, 476–488.
17. Liang, X., Nguyen, M., and Che, H. Wimpy or brawny cores: A throughput perspective. *Journal of Parallel and Distributed Computing 73*, 10 (Oct. 2013), 1351–1361.
18. Meisner, D., Sadler, C.M., Barroso, L.A., Weber, W.D., and Wenisch, T.F. Power management of online data-intensive services. In *Proceedings of the 38th Annual International Symposium on Computer Architecture* (San Jose, CA, June 4–8). ACM Press, New York, 2011, 319–330.
19. Meisner, D. and Wenisch, T.F. Does low-power design imply energy efficiency for data centers? In *Proceedings of the 17th IEEE/ACM International Symposium on Low-Power Electronics and Design* (Fukuoka, Japan, Aug. 1–3). IEEE Press, Piscataway, NJ, 2011, 109–114.
20. Novakovic, S., Daglis, A., Bugnion, E., Falsafi, B., and Grot, B. Scale-Out NUMA. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, UT, Mar. 1–5). ACM Press, New York, 2014, 3–18.
21. Ousterhout, J., Agrawal, P. et al. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. *SIGOPS Operating Systems Review 43*, 4 (Jan. 2010), 92–105.
22. Reddi, V.J., Lee, B.C., Chilimbi, T., and Vaid, K. Web search using mobile cores: Quantifying and mitigating the price of efficiency. In *Proceedings of the 37th IEEE/ACM International Symposium on Computer Architecture* (Saint-Malo, France, June 19–23). ACM Press, New York, 2010, 314–325.
23. Sprangle, E. and Carmean, D. Increasing processor performance by implementing deeper pipelines. In *Proceedings of the International Symposium on Computer Architecture* (Anchorage, AK, May 25–29). IEEE Press, 2002, 25–34.
24. Trivedi, K. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications, Second Edition.* John Wiley & Sons, Inc., New York, 2002.

Christina Delimitrou (delimitrou@cornell.edu) is an assistant professor and the John and Norma Balen Sesquicentennial Faculty Fellow in the Department of Electrical and Computer Engineering at Cornell University, Ithaca, NY, USA.

Christos Kozyrakis (kozyraki@stanford.edu) is a professor in the Departments of Electrical Engineering and Computer Science at Stanford University, Stanford, CA, USA.