Research Statement

Benjamin Berg

Abstract

My research interests are in the area of *stochastic performance modeling* and *scheduling*, with an emphasis on the use of stochastic models to optimize real-world computer systems. My work proposes the first stochastic models of parallelizable jobs running in multicore or multisever systems. I derive the first the optimal allocation policies under these stochastic models, and use these policies to improve on the state-of-the-art policies used in databases. I have also worked on caching in large-scale web services, and I have collaborated closely with Facebook to improve their production caching systems.

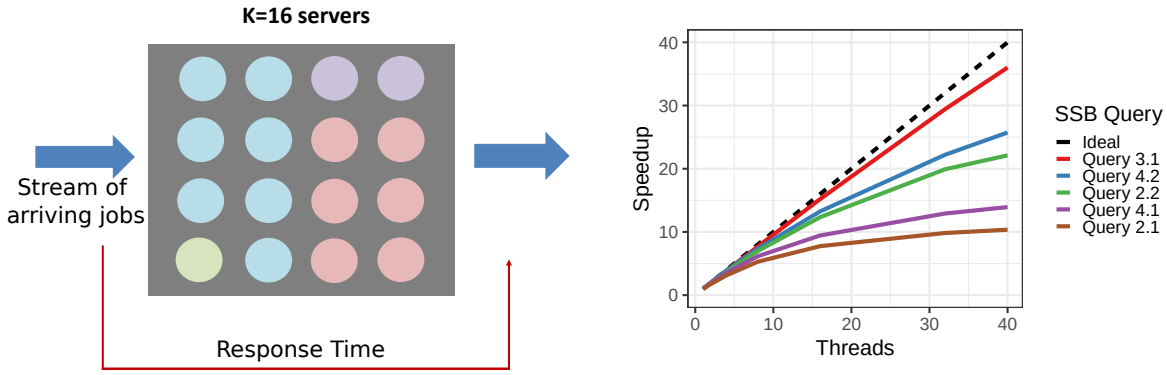
1 My Research Philosophy

The Importance of Resource Allocation. Many problems in the design of modern computer systems are fundamentally *resource allocation problems*, where one must decide how to effectively share a set of hardware resources between multiple simultaneous users. This question has spawned entire subfields of systems research from operating systems to networking to computer architecture. For many years, the need for improved resource allocation policies has been masked by a steady improvement in the underlying hardware. However, as has been well-documented, we are reaching the physical limits of the hardware improvements that have driven progress over the last 50 years [18]. As a result, the development of algorithms derived from new theoretical models is poised to become a major source of performance improvement in the coming years [13]. Because system designers will seek improved performance from a fixed set of hardware resources (i.e. a fixed number of transistors) the need to employ good resource allocation policies will become more acute.

The Gap Between Theory and Practice. Resource allocation problems have been studied extensively by the theoretical computer science (TCS) community. However, theoretical results have traditionally taken a worst-case approach to analyzing these problems, assuming that system workloads are chosen adversarially. This has led the TCS community to derive strong lower bounds or hardness results for a variety of resource allocation problems. Meanwhile, systems researchers have successfully developed performant systems by relying on heuristic allocation policies that vastly outperform the pessimistic worst-case theoretical bounds. We can therefore conclude that the adversarial examples limiting the performance of allocation policies in the worst case are not common enough to govern the performance of real-world systems.

A Stochastic Approach to Bridge Theory and Practice. To close the gap between resource allocation in theory and in practice, my research takes a *stochastic modeling* approach and assumes that workloads are drawn from distributions. These distributions more closely match the kinds of workloads seen in real-world systems and reflect the observation that worst-case scenarios are uncommon in practice. I have developed the first stochastic models of resource allocation for parallelizable jobs, and using these models I have derived a variety of resource allocation policies that significantly improve performance over existing policies from the theoretical literature, as well as the heuristic policies used in practice.

The crux of my research philosophy is that there is a need for research directly at the interface between computer science theory and systems. I therefore work on theoretical problems as well as system design and implementation to both validate my theoretical models and maximize the impact of my research on real-world systems. My style of research is also well-served by collaborating with industry partners to learn about current problems and deploy research ideas at scale. In the next two sections, I will discuss how I have applied my research approach to the problems of resource allocation for parallelizable jobs as well as caching in large-scale web services.



(a) Allocating servers to jobs

(b) Typical speedup functions

Figure 1: Our model of parallelizable jobs running in a multi-server system. A job's response time is the time from when a job arrives to the system until the job is completed. An allocation policy decides, at every moment in time, how many servers to allocate to each job in order to minimize the mean response time across jobs. Job allocations are denoted by the different server colors. Each job has a corresponding *speedup function* $s(k)$. Given an allocation of k servers, a job is worked on at a rate of $s(k)$. Figure 1b shows speedup functions of several queries from the Star Schema Benchmark [16] in the NoisePage database [2].

2 Scheduling Parallelizable Jobs

Modern systems, from mobile processors to warehouse-scale computers, are now purpose-built to exploit parallelism in myriad ways. Accordingly, modern workloads are increasingly composed of parallelizable jobs which are designed to complete more quickly when run on a higher number of cores or servers. Unfortunately, it is often the case that jobs receive a *sublinear speedup* from running on additional servers as shown in Figure 1. My research asks the following question:

> **Given a stream of jobs with different inherent sizes (inherent work) and different speedup functions, how should one allocate cores or servers to jobs in order to minimize metrics such as the mean response time across jobs?**

All prior theoretical work has addressed this problem through the lens of worst-case analysis, where [14] established strong lower bounds showing that it is *impossible* to perform within a constant factor of the optimal policy. My research models job sizes, arrival times, and speedup functions as being drawn from underlying distributions rather than being generated by an adversary. This approach more closely reflects the reality of modern workloads and eliminates the worst-case lower bounds on achievable performance.

The Tradeoffs in Scheduling Parallelizable Jobs

The challenge in deriving good allocation policies arises from the counterintuitive interaction between several competing goals that arise in the system. My research has revealed that there are several tradeoffs that must be weighed simultaneously. The central tradeoff in the system is that an allocation policy must choose between completing an individual job more quickly, and operating the system at a higher level of efficiency. Giving a large allocation to a single job will cause that job to complete more quickly, but it is also wasteful since jobs receive a diminishing marginal benefit from running on additional cores or servers. Although it is not immediately obvious, we can show that if all jobs look identical in terms of size and speedup function, a policy should should maximize system efficiency rather than favoring any individual job. The story becomes more complex, however, when jobs do not all look the same. If some jobs are more parallelizable than others, the *present* system efficiency can be maximized by favoring the more parallelizable jobs. However, this can lead to *future* inefficiency if the system is left with a small number of the less parallelizable jobs. Hence, it may be worthwhile to operate with lower efficiency in the present to preserve the future efficiency of the system. Similarly, if some jobs are known to have smaller sizes than others, a scheduling policy may want to favor smaller jobs to reduce mean response time. However, one must decide how heavily to favor short jobs,

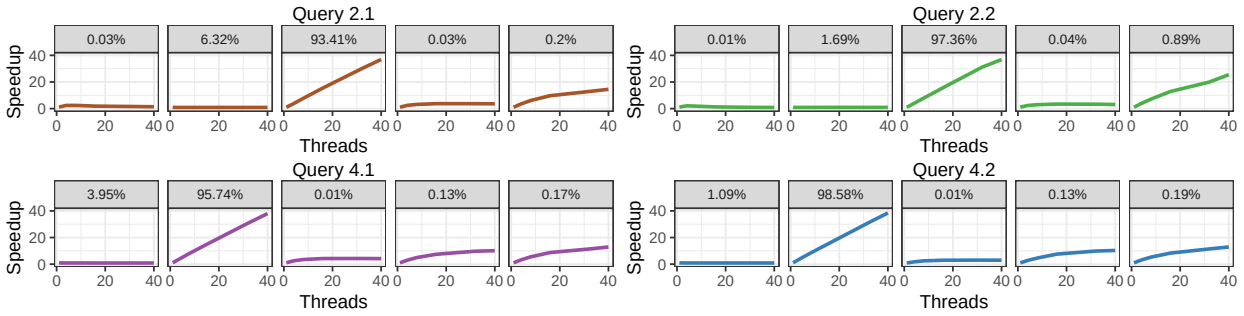


Figure 2: Speedup functions for each phase of four queries from the Star Schema Benchmark. Queries were executed using the NoisePage database [2]. Phases are either elastic (highly parallelizable) or inelastic (highly sequential). The percentages denote the fraction of time spent in each phase when the query was run on a single core. Despite the queries spending most of their time in elastic phases, the overall speedup function of each query is highly sublinear due to Amdahl's law.

since favoring an individual job decreases overall system efficiency. My work on these tradeoffs is described in greater detail in the following sections.

Speedup vs. Efficiency. From the perspective of a single job, it is always better to run on more cores or servers and receive the maximum possible speedup. However, from the perspective of the other jobs in the system, such an allocation may seem very wasteful if jobs experience diminishing marginal benefits from receiving additional cores or servers. In [4], we define a class of allocation policies, called GREEDY, composed of policies that choose a maximally efficient allocation at every moment in time. When jobs all follow the *same* speedup function and the *same* exponential size distribution, we can show that a GREEDY policy minimizes mean response time. Specifically, when all jobs are "equal", a GREEDY policy called EQUI maximizes efficiency by dividing resources equally among all jobs in the system.

Deferring Parallelizable Work. Of course, in real systems, all jobs are generally not "equal". For example, if we consider a typical datacenter workload, jobs can vary widely in terms of their speedup functions, with some jobs being highly parallelizable and other jobs being highly sequential. In [4], we find that the best policy in the GREEDY class is GREEDY*, a policy that *defers parallelizable work* by favoring less parallelizable jobs. While this policy is not optimal in general, it performs well in simulation. In [5], we find sufficient conditions for the optimality of GREEDY*. Specifically, we consider the case where workloads are composed of a mixture of large *elastic* jobs which receive an ideal speedup, and smaller *inelastic* jobs which can run on at most one server. In this case, GREEDY* is equivalent to a policy called *Inelastic First* (IF) which gives strict priority to inelastic jobs. If job sizes continue to follow an exponential distribution, IF is optimal with respect to mean response time. This defies the intuition that more parallelizable jobs, which can effectively utilize resources in parallel, should always receive favorable allocations.

Favoring Short Jobs. When job sizes are known to this system, it is generally the case that favoring jobs with smaller inherent sizes tends to reduce mean response time. In the single server case, for instance, the policy which gives strict priority to the job with the *shortest remaining processing time* (SRPT) is optimal for any sequence of arrivals. However, in the case of parallelizable jobs in a many-server system, it turns out that using SRPT can have a disastrous effect on system efficiency. Given a set of jobs with known, generally distributed inherent sizes, we derive an exact closed-form for the optimal policy, which we call *high-efficiency SRPT* (heSRPT) [6]. The heSRPT policy favors short jobs *and* maintains the overall efficiency of the system. The key is to find the point at which the marginal benefit of increasing the priority of short jobs is equal to the marginal loss in system efficiency. This work provides a generalization of a ubiquitous result, the optimality of SRPT, to a multi-server context. It also underscores the importance of building systems that can accurately track, in real time, the remaining inherent sizes of their jobs, since this information can both greatly improve response times and drastically change the form of the optimal allocation policy.

Implementing Resource Allocation in Databases

Databases are a natural application area for my theoretical results, since a stream of parallelizable queries is often required to share a limited set of underlying cores. Database queries are particularly interesting because a single query is generally broken into a series of alternating elastic and inelastic operations. For example, a single query is highly parallelizable while executing a table scan, and highly sequential when executing a table join. Hence, while database queries exhibit speedup functions as shown in Figure 1b, these speedup functions are measuring each query's *average* speedup over its entire lifetime. Figure 2, on the other hand, shows how a query's parallelizability changes over time as it moves through distinct phases [16].

Unfortunately, the state-of-the-art in modern databases to use a very simple first-come-first-served (FCFS) allocation policy which is highly suboptimal. To improve on the state-of-the-art in database scheduling, I have extended my theoretical results on the IF allocation policy to apply to the case where jobs consist of multiple phases [7]. I have also worked to implement this phase-aware IF allocation policy in the NoisePage database [2].

Implementing IF in NoisePage has required addressing several systems challenges. First, I had to modify the database execution engine to separate parallelizable and non-parallelizable routines into disjoint elastic and inelastic execution steps. Then, I built a new scheduler that could consume metadata about the type and duration of each execution step. My design leverages the idea of Morsel-driven parallelism [12] to preempt queries efficiently and give priority to queries in inelastic phases. This design also allows the scheduler to include cache and NUMA locality information when deciding how to map queries to available cores.

My work illustrates the importance of considering how to allocate resources between simultaneously running queries to reduce query latency. Traditionally, the database community has improved performance by optimizing individual query execution or increasing system throughput. My research shows that these systems stand to drastically improve performance by implementing improved resource allocation policies.

3 Caching

Another component of my research has been the development of caching systems for large-scale web services, where I have collaborated extensively with industry to facilitate the deployment of new caching systems. Caches appear at nearly every level of modern web services and are essential for allowing users to quickly access data which is stored in massive distributed databases. Consider, for example, the Facebook social graph which as of 2015 had 1.39B users connected by 400B edges [10]. A single user's request may load hundreds of edges from the social graph, and thus massive caches are required to serve pages efficiently [9]. This begs the question of which items to store in such a cache, and how to design cache hierarchies that achieve the best performance.

Although caching feels like a vastly different problem from scheduling, both problems require deciding how to effectively share a constrained resource. In particular, because most systems exhibit diminishing marginal increases in hit ratio from being allocated additional cache space, a system's performance as a function of its available cache space can vary much like the performance of a parallelizable job with a concave speedup function. This observation led me to work on the Robinhood caching system [8], which dynamically allocates cache space between different backend systems in a web service architecture to minimize the P99 tail latency of user requests. By dynamically allocating cache space, Robinhood can cause user request to meet their target P99 latency 99.7% of the time. I also worked on the Kangaroo caching system [15], which develops a new technique for using flash devices to build large capacity caches. By optimizing the pattern of writes to the underlying flash device, Kangaroo can decrease cache miss ratios by roughly 20%.

In addition to developing new caching algorithms and system designs, I have collaborate closely with industry to maximize the impact of my research. Specifically, I found that modern caching systems presented many barriers to widespread uptake of new ideas in cache design. Because the dozens of caching systems within a typical production environment have traditionally been developed as separate, specialized systems, deploying even a single new caching feature can require duplicated implementations in each system. This observation led me to work with Facebook to develop, analyze, and open-source CacheLib [3, 1], a general-purpose caching engine that provides a library of reusable components for building production-ready caching systems. CacheLib creates a pathway for ideas from the research community to migrate to production systems. If a new feature is built as a CacheLib component, existing systems that use CacheLib can access

this feature via a simple CacheLib version bump. The result is that caching researchers now have a flexible baseline cache implementation with which to evaluate their ideas, and new caching research can have a more immediate impact on production systems. The power of this development model was illustrated during my work on Kangaroo, which was built using CacheLib and then immediately deployed into Facebook's production environment.

4 Future Research

My research thus far has focused on optimally allocating a fixed set of resources with the goal of minimizing response times. However, my work really represents a first step in understanding the dynamics of a larger class of resource allocation problems. By considering different metrics, different underlying hardware, and additional resource constraints, I plan to build on the basic tenets of my thesis research and pursue several new directions for future research.

4.1 Allocation in Heterogeneous Systems

My research has extensively considered cases where the workload of arriving jobs are heterogeneous with respect to their sizes and speedup functions. However, in a variety of systems, it is also the case that the underlying hardware is heterogeneous. For example, system architects have identified the potential benefit of building multicore systems with a mix of large, fast cores and small, slow cores [11]. This has led to the development of heterogeneous architectures such as the ARM big.LITTLE, however these systems generally defer decisions about thread placement to the user. Similar problems also arise in data centers, where server SKUs and configurations can vary widely. Here, an allocation policy must decide not only how many servers to give a particular job, but what server speeds, amount of memory, and hardware accelerators will be associated with a particular allocation. I propose to model and analyze these systems, and then build OS and data center schedulers that can fully exploit the heterogeneous hardware.

4.2 Allocation with Budget Constraints

Although my research has assumed that there is a fixed set of available cores or servers, there are many important settings where the amount of available processing power might be subject to change over time. These changes in the available cores or servers are typically driven by the existence of additional *budget constraints*. For example, modern multicore processors can use dynamic voltage and frequency scaling to run at faster speeds, but these speed boosts must be carefully controlled to prevent overheating. Data center operators are often tempted to turn servers off rather than allocate them to save power. And applications which leverage cloud computing must decide how to scale their virtual machine deployments up and down to achieve good performance without spending too much money. In these cases, additional budget constraints force allocation policies to consider the tradeoff between increased cost and increased performance. I propose to derive and implement allocation policies that can optimize system performance subject to a fixed power or monetary budget.

4.3 Allocation for Model Training

The importance of efficiently training and serving machine learning models will only continue to increase in the coming years. When training a machine learning model, a job is typically run until it reaches a high degree of accuracy. It is generally thought to be hard to predict how much training will be required for a model to reach the desired accuracy. Hence, one might train several different models in parallel (with different hyperparameters, say) until some model reaches a predetermined accuracy threshold. Given a set of training nodes (or a fixed dollar budget to spend, see above), this begs the question of how one should allocate resources to models across the entire hyperparameter space. Here, in addition to the tradeoff between individual speedup and system efficiency, one must balance the classic tradeoff between exploration and exploitation. I propose to adapt the insights from my existing research to generate better allocation policies for machine learning training workloads.

References

[1] Cachelib: Pluggable caching engine to build and scale high performance cache services. `https://cachelib.org/`.

[2] NoisePage - The Self-Driving Database Management System. `https://noise.page`.

[3] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, et al. The cachelib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 753–768, 2020.

[4] Benjamin Berg, J.P. Dorsman, and M. Harchol-Balter. Towards optimality in parallel scheduling. *ACM POMACS*, 1(2), 2018.

[5] Benjamin Berg, Mor Harchol-Balter, Benjamin Moseley, Weina Wang, and Justin Whitehouse. Optimal resource allocation for elastic and inelastic jobs. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 75–87, 2020.

[6] Benjamin Berg, Rein Vesilo, and Mor Harchol-Balter. heSRPT: Parallel scheduling to minimize mean slowdown. *Performance Evaluation*, 144:102–147, 2020.

[7] Benjamin Berg, Rein Vesilo, and Mor Harchol-Balter. The case for phase-aware scheduling of parallelizable jobs. *To Appear: Performance Evaluation*, 2021.

[8] Daniel S Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. Robinhood: Tail latency aware caching–dynamic reallocation from cache-rich to cache-poor. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 195–212, 2018.

[9] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. *TAO*: Facebook's distributed data store for the social graph. In *2013 USENIX Annual Technical Conference (USENIXATC 13)*, pages 49–60, 2013.

[10] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[11] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41:33–38, 2008.

[12] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 743–754, 2014.

[13] Charles E Leiserson, Neil C Thompson, Joel S Emer, Bradley C Kuszmaul, Butler W Lampson, Daniel Sanchez, and Tao B Schardl. There's plenty of room at the top: What will drive computer performance after moore's law? *Science*, 368(6495), 2020.

[14] Stefano Leonardi and Danny Raz. Approximating total flow time on parallel machines. *Journal of Computer and System Sciences*, 73(6):875–891, 2007.

[15] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel Berger, Nathan Beckmann, and Gregory R Ganger. Kangaroo: Caching billions of tiny objects on flash. In *To Appear: Symposium on Operating Systems Principles (SOSP)*, 2021.

[16] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. *The Star Schema Benchmark and Augmented Fact Table Indexing*, pages 237—-252. 2009.

[17] Dennis M Ritchie and Ken Thompson. The UNIX time-sharing system. *Bell System Technical Journal*, 57(6):1905–1929, 1978.

[18] Thomas N Theis and H-S Philip Wong. The end of moore's law: A new beginning for information technology. *Computing in Science & Engineering*, 19(2):41–50, 2017.