Microeconomic Models for Managing Shared Datacenters

by

Qiuyun Llull

Department of Electrical and Computer Engineering
Duke University

Date: ________________________
Approved:

____
Benjamin C. Lee, Supervisor

____
Jeffrey S. Chase

____
Alvin R. Lebeck

____
Gabriel H. Loh

____
Daniel J. Sorin

Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Electrical and Computer Engineering in the Graduate School of Duke University
2017

<u>Abstract</u>

Microeconomic Models for Managing Shared Datacenters

by

Qiuyun Llull

Department of Electrical and Computer Engineering
Duke University

Date: ____________________________

Approved:

____
Benjamin C. Lee, Supervisor

____
Jeffrey S. Chase

____
Alvin R. Lebeck

____
Gabriel H. Loh

____
Daniel J. Sorin

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Electrical and Computer Engineering
in the Graduate School of Duke University
2017

Abstract

As demands for users' applications' data increase, the world's computing platforms are moving towards more capable machines – servers and warehouse-scale datacenters. Diverse users share datacenters for complex computation and compete for shared resources. In some systems, such as public clouds where users pay for reserved hardware, management policies pursue performance goals. In contrast, private systems consist of users who voluntarily combine their resources and subscribe to a common management policy. These users reserve the right to opt-out from shared systems if resources are managed poorly. The system management framework needs to ensure fairness among strategic users, encouraging users to participate while guaranteeing individual performance and preserving the system's performance. Microeconomic models are well suited for studying individual behavior and the allocation of scarce resources. In this thesis, we present three pieces of work on task colocation, resource allocation and task scheduling problems to demonstrate the effectiveness of a microeconomic approach.

Colocating applications on shared hardware (i.e., chip-multiprocessors) improves server utilization but introduces resource contention into the memory subsystem. In the first work, we design a colocation framework based on cooperative game theory to manage shared resource contention. Our framework uses a recommendation system to predict individual applications preferences for colocated tasks. It then uses these predictions to drive novel colocation mechanisms to guarantee user fairness and preserve system performance. Attractive system outcomes encourage strategic users to participate in the datacenter.

Processor allocations are inefficient when they are based on static reservations because reservations are often conservative; users rarely know their application's needs across time, especially when applications have complex phase behavior. In the second work, we propose a fast, lightweight performance prediction framework to help users capture their phase behaviors in parallel applications. We design a dynamic and distributed core allocation framework so that users can trade resources for better efficiency based on predicted performance. Our management framework provides efficient allocations and game-theoretic fairness guarantees.

In the last work, we characterize applications' sensitivity to non-uniform memory access (NUMA) in big memory servers. We develop performance and energy models for communication costs in a blade server. We use this model to perform case studies on NUMA-aware scheduling policies and task queue management. Our parameterized models lay the foundation for coordinated design of scheduling policies and hardware configurations. This method can be further used to design locality-aware schedulers with microeconomic models, e.g., dynamic pricing strategies for city parking.

For my parents, friends and teachers — people who helped me to come this far.

Contents

List of Tables

List of Figures

Acknowledgements

The journey to this PhD dissertation is long and challenging. Without many people, this dissertation would never have materialized. First of all, I want to thank my adviser, Dr. Benjamin Lee, for giving me the opportunity to work as a Ph.D. student in a great research lab. In the five years we have worked together, he has dedicated many hours to helping me improve my technical research and professional abilities, and our discussions have refined my soft skills and career direction. Dr. Lee was always confident in the quality of our research and sought out the best venues; this dissertation consists of our research that was accepted into a variety of competitive conferences.

The members of my exam committee have all been tremendously helpful during each of my milestones: Dr. Daniel Sorin and Dr. Alvin Lebeck, for their expertise in computer architecture, especially memory systems to help with defining realistic system settings; Dr. Jeffrey Chase, for his expertise in large-scale distributed systems and great research intuition in resource management framework design; and Dr. Gabriel Loh, for his suggestions in performance modeling and project direction as a whole. It is my great honor to have these top researchers in the field on my committee and to be able to discuss my work in depth with them.

This thesis is an interdisciplinary study between computer architecture and algorithmic economics. Without the wisdom in algorithms and economics, it would never have been complete. I would like to extend my thanks to Dr. Vincent Conitzer, for his valuable input on the micro-economic models and game theory part of my project; two Ph.D. students in algorithms Dr. Janardhan Kulkarni and Brandon Fain for various theory discussions we had on my projects.

I would also like to express my thanks to the professors in Duke Computer Architecture group, especially Dr. Daniel Sorin, Dr. Alvin Lebeck and Dr. Andrew Hilton. Throughout my study in graduate school, I took various computer architecture and system related classes from them to enhance my CS fundamentals; they gave prompt research feedback during my conference practice talks; they led weekly reading group sessions to keep students updated with state-of-art research in the field; and they are always willing to give career advice and connect me to internships or full-time opportunities.

Getting a PhD is a combination of professional development and emotional training. I would like to thank many important people that accompanied me in this unforgettable journey and helped me reach the finishing line: Dr. Marisabel Guevara and Dr. Songchun Fan for their valuable advice on graduate school life, career development and uncountable discussions on my projects which turn to be extremely helpful; Seyed Zahedi for his dedication in our collaborative projects and his incredible knowledge in game theory; my numerous brilliant and talented labmates, Luwa Matthews, Tamara Lehman, George Mappouras, Pengfei Zheng, Ziqiang Huang and Dr. Blake Hechtman, who added the much-needed fun element to my Ph.D. life and form priceless lifelong friendship. And finally, last but by no means least, I would like to thank my husband, my parents, my family and my friends for their continuous support, love and advice throughout this time, regardless of which stage of graduate school I was in. Without their love and encouragement, I would not have had the strength and courage to reach the end.

1 Introduction

As demands for users' applications' data increase, the world's computing platforms are moving towards more capable machines – servers and warehouse-scale datacenters. Datacenters are large-scale systems equipped with racks of servers. Internet service companies build datacenters to provide web services including search engines, social media, video streaming, email, office tools, and remote storage. Public cloud providers such as Amazon Web Service, Microsoft Azure and Google Cloud rent bare-metal servers or virtual machines along with management platforms to companies and individuals. Private datacenters, such as university department clusters or a company's internal development clusters, provide powerful computing platforms for researchers and developers. Many efforts are dedicated to improve datacenter performance. In hardware, companies like Intel and AMD continuously improve individual servers' capabilities with technology advances, for example, by increasing number of CPU cores, putting accelerators on-chip, adding more memory and building faster storage. In software, system management frameworks deploy resource management policies to improve server utilization and service quality with a much shorter production cycle.

Resource management policies include resource allocation, task scheduling and task colocation. Different techniques are used in different system settings. In some circumstances, users provide resource requirements, such as number of cores, amount of memory or number of virtual machines. This usually happens in public cloud services where users pay for a certain time frame for a fixed amount of resource. In this case, the system manager uses bin packing [38], load balancing [62, 67], live migration, or sophisticated optimization techniques to improve server utilization while monitoring VM service quality. In other circumstances, where applications are submitted to a shared cluster to perform computation, system management software should leverage performance profiling and prediction techniques [27, 70, 113] to infer applications' resource requirements and make resource management decisions based on these performance implications [28, 64].

Predicting datacenter application resource requirements for efficient system management is challenging. First, the prediction has to be fast. Applications are numerous and diverse. Simple heuristics can be effective if applications are similar. Unfortunately, even applications within the same type can behave significantly differently. Batch jobs may be compute intensive, memory intensive or I/O intensive; graph applications may have skewed data sets that change communication patterns. Second, performance prediction has to consider the effects on hardware. For example, colocated applications on the same processor might cause resource contention in the memory subsystem, which impacts performance; servers with large memory have non-uniform memory access that may cause slow response time on critical tasks and may affect the overall application performance. Finally, performance models have to be simple. Finding the optimal solution for resource allocation and scheduling problems is generally combinatorial. Greedy and heuristic strategies can help reduce the problem dimension but require performance models as input. Therefore, keeping a simple, light-weighted model is critical to keep the solution space small and to make fast management decisions.

System performance is not the only concern for system managers. Conventional wisdom assumes that users must share and policies need only optimize for performance. Such performance goals are suitable for public systems that deliver hardware for which users have paid. In contrast, private systems consist of users who voluntarily combine their resources and subscribe to a common management policy. These users also reserve the right to withdraw from the system if resources are managed poorly. Therefore, privately shared systems must manage resources fairly to encourage participation and guard against strategic behavior [120]. Real-world users are selfish and rational [9, 24], an observation that has motivated numerous game-theoretic perspectives on systems management [16, 32, 36, 85, 119]. Neglecting users' preferences or fairness induces strategic behavior. Users may circumvent policies or break away from shared clusters, redeploying hardware to form smaller, separate systems [36, 82]. Guaranteeing fairness for shared users in system management policies can address these challenges, ensuring system integrity and cluster efficiency.

Microeconomics models are the best practice for studying individual behavior in making decisions regarding the allocation of scarce resources [101]. Market theory is a common microeconomic theory that allocates scarce resources to shared users by establishing a competitive market price. Game theory is another major method in modeling competing behaviors of strategic users; it is widely used in auctions, fair division, social network formation, voting systems, etc. These well-established microeconomic theories provide the foundation for designing system management policies that target shared datacenters with strategic users. In this dissertation, we will use three pieces of work to demonstrate the work flow and effectiveness of this approach. The focus of this thesis is not about exploring various perspectives of microeconomic theories or inventing new theories. Rather, it will focus on formalizing realistic system settings for resource management problems in shared datacenters, designing concise and accurate performance models to drive management policies, and adapting microeconomic theories to design efficient and fair system management frameworks.

1.1 Organization and Overview of the Dissertation

This thesis aims to improve datacenter applications' performance under the constraint of guaranteeing user fairness. We present three pieces of work that investigate different system management policies, from task colocation to resource allocation to task scheduling, and coordinate the policy design with different performance modeling techniques, from fitting mathematical utility functions to training machine learning models to deploying cross-level simulators.

This thesis is an interdisciplinary study of computer architecture and microeconomics. Compared to prior works in system management and policy design, the key innovation here is exploring theories in the field of microeconomics for management mechanisms that can guarantee fairness while preserving good system performance and adapting these theories to current datacenter resource management problems. At the same time, we show designs for performance models that consider both the underlying hardware and application behavior to drive management mechanisms. This combined approach can better utilize hardware, improve system efficiency and provide attractive outcomes for users in shared systems.

1.1.1 Cooperative Games to Manage Colocation Contention

Datacenter system managers colocate tasks to improve server utilization and energy-proportionality, but often introduce resource contention for shared hardware. In the first work, we discover that traditional performance-centric colocation mechanisms can reduce resource contention but often lead to strategic behavior for shared users in private datacenters. Such strategic behavior harms system stability and cluster integrity.

Chapter 2 presents the first study on fair task colocation — `Cooper`, a colocation framework that uses cooperative game theory [76, 80] to formalize interactions between strategic users and to enable fair task colocation. `Cooper` builds user preference prediction models with collaborative filtering techniques [94] and designs three matching mechanisms to finds stable matches with predicted preferences. Its colocations satisfy preferences and encourage strategic users to participate in shared systems. We find that given `Cooper`'s colocations, users' performance penalties are strongly correlated to their contributions to contention, which is fair according to cooperative game theory. Finally, its colocations preserve system performance; we observe performance within 5% of prior heuristics. (Chapter 2)

1.1.2 Market Mechanism to Allocate Cores for Parallel Applications

In the second work, we consider another datacenter resource management component — resource allocation. Instead of guaranteeing a single application's performance and fairness, we consider scenarios in which each user holds multiple applications and the total performance of all of her applications is the goal. Users share machines in a federated datacenter; each user has an endowment of resources. Users can trade CPU resources from one application to supply another application that will benefit more at the current time.

In Chapter 3, we present a performance prediction framework for users' parallel applications with large data sets. We use Amdahl's Law [8] and Karp-Flatt [43] to estimate each application's utility from any core allocation. We validate our performance model with real system measurements of benchmarks from Apache Spark and predict execution time for varied core counts and dataset sizes with an average of 15% error. We propose a Fisher Market that leverages our lightweight performance predictor along with a distributed bidding mechanism to find a market equilibrium [121]. We discover that resulting core allocations outperform state-of-art performance-centric mechanisms with collaborative filtering predictions [94]. More importantly, they provide strong game-theoretical guarantees on envy-freeness, sharing incentives and Pareto-efficiency. (Chapter 3)

1.1.3 Performance and Energy Models for Blade Servers

The previous two works demonstrate the coordinated approach for designing performance models and system management policies In some scenarios, like task scheduling with multiple queues, closed form utility functions can only model simple queue structures (i.e., M/M/1 or M/M/K queues). Complex hardware poses challenges in building fast and accurate performance models for task scheduling. In this work, we investigate datacenter big memory servers — blade servers — which disaggregate memory across multiple blades to provide dense memory capacity for big data applications [60, 61]. Larger memory capacities cause deeper non-uniform memory access (NUMA), which complicates NUMA-aware scheduling policy design [18, 26].

In Chapter 4, we derive technology and architectural models to estimate communication delay and energy for blade servers. Additionally, we propose a multi-stage simulation method with detailed architecture and discrete-event simulators to estimate application performance and communication channel energy costs. These models permit new case studies in delay scheduling to mitigate NUMA and improve the energy-efficiency of data movement. Preliminary results from our case study show that these models can help researchers coordinate NUMA mitigation and task queueing dynamics. We find that judiciously permitting NUMA reduces queueing time, which benefits throughput, latency and energy-efficiency for datacenter workloads like Apache Spark[118]. These findings highlight blade servers' strengths and opportunities when building distributed shared memory machines for data analytics. Finally, our models lay the foundation for estimating communication costs within memory systems with flexible interconnection technology and topology. Future studies in task scheduling policy design can benefit from these models to optimize for system goals, like performance, fairness, energy-efficiency. (Chapter 4)

1.1.4 Key Contributions of the Thesis

In summary, we make the following contributions in this thesis.

- We define fair task colocation as fair attribution of costs, satisfied user preferences, and stable colocations. We propose the first study on fair colocation with cooperative games and demonstrate that our task colocation framework can provide these system desiderata while preserving system performance. We prototype the task colocation framework in a local cluster with the following system components: system profiler, preference predictor, colocation policies, action recommender and job dispatcher.

- We propose a performance model and a lightweight prediction framework based on Amdahl's Law and Karp-Flatt metric to model parallel applications' performance. We validate the performance model by real system measurements of big data applications. We propose a Fisher Market solution to the multi-user resource allocation problem and a distributed bidding mechanism that leverages the performance models to find a market equilibrium. We show that the resource allocation mechanism can provide game-theoretic desiderata while preserving system performance.

- We provide a detailed model of communication costs for a modern big memory server in latency, power, and energy. We characterize applications' sensitivity to non-uniform memory access with architectural models, and perform a coordinated study of NUMA-aware scheduling policy design and task queueing dynamics with system simulation.

2

Task Colocation with Cooperative Games

Modern datacenters, with their increasingly parallel computation and increasingly capable machines, colocate small tasks on big servers. Colocating multiple tasks on each server increases efficiency and energy proportionality [15] but introduces contention for shared resources such as last-level cache capacity and memory bandwidth [96, 99, 122].

In a privately shared system, where strategic users have options to bypass the management policy or opt out from sharing, colocation contention can cause strategic behavior and harm shared system integrity. For example, in a computer science department, three professors (A, B, C) are sharing a cluster equally and working towards a conference deadline. Professor A's applications are mostly memory-intensive. Professor B and C's applications are similar light-weighted applications. If the cluster manager colocates professor A and B's applications together, professor B's applications will suffer great performance losses. Therefore, strategic user (professor) B would migrate his applications to colocate with professor C's application to gain performance. Professor B and C have no incentive to colocate with professor A. If many strategic users are dissatisfied with their colocation assignment and migrate their applications, the system suffers from non-deterministic performance. If many users opt out and form small shared systems to improve their performance, the system loses its integrity.

One of the many challenges to manage shared systems is ensuring fairness among strategic users and incentivize them to participate in such shared environment. Prior work focuses on colocation policies that improves system performance. It manages colocation performance by controlling contention, which depends on the tasks colocated. Because finding the best colocations requires combinatorial optimization, practical heuristics often colocate tasks if performance penalties are tolerable [27, 70] or use architectural insights to pair applications with complementary resource demands [96, 99].

Conventional wisdom assumes that users must colocate and policies need only mitigate contention. Such performance goals are suitable for public systems that deliver hardware for which users have paid. In contrast, private systems consist of users who voluntarily combine their resources and subscribe to a common management policy. These users also reserve the right to withdraw from the system if resources are managed poorly.

Therefore, privately shared systems must manage resources fairly to encourage participation and guard against strategic behavior. Real-world users are selfish and rational, an observation that has motivated numerous game-theoretic perspectives on systems management [16, 32, 36, 85, 119].

We pursue our system desiderata with cooperative games. Game theory is a framework for analyzing outcomes from strategic behavior. Cooperative games describe how agents' interactions dictate shared outcomes. Such games are well suited for colocation as interference between tasks dictates performance penalties. Cooperative games build a foundation for fair colocation, which encourages strategic users to share. The following summarizes our contributions:

- **Fair Colocation.** We present the case for three desiderata from colocation: (i) fair attribution such that more contentious users incur larger penalties, (ii) satisfied preferences such that more users colocate with preferred co-runners, (iii) stable colocations such that fewer users break away from the shared system.

- **Cooperative Games.** We formalize the colocation game in which users share hardware and contention causes performance losses. When assigning colocations, the game accommodates users' preferences for co-runners. The game's equilibrium produces fair and stable systems.

- **Colocation Framework.** We present `Cooper`, a cooperative game that predicts preferences and colocates tasks. It adapts stable matching algorithms to the colocation problem. It then assesses colocations and recommends strategic actions for users.

- **Multiprocessor Evaluation.** We evaluate Spark and PARSEC jobs that share chip multiprocessors. We show that `Cooper`'s colocations are fair as jobs' performance losses increase with their demands for memory. Colocations also satisfy users' preferences, which encourages sharing. `Cooper` performs within 5% of prior heuristics.

2.1 Case for Fair Colocation

For the first time, we present the case for fair colocation. In economics, fairness is the equal treatment of equals and the unequal treatment of unequals in proportion to their relevant differences [11, 76]. *We say colocations are fair when similar tasks suffer similar performance losses.* We argue that, when tasks are dissimilar, the relevant differentiator is contentiousness. Thus, users' performance losses from colocation should increase with their contributions to contention.

We approach fair colocation from a game-theoretic perspective, which formally describes strategic situations, because systems are often shared by strategic users. Cooperative game theory prescribes the fair division of costs that arise from interactions between strategic agents [80]. Solutions to these games reconcile agents' divergent preferences to produce stable outcomes [35, 51]. We use such theories to design colocation policies and build frameworks to manage datacater colocation contention.

2.1.1 Desirable System Desiderata

We manage datacenters that colocate strategic users and their tasks on chip multiprocessors. We define *strategic users* as those who selfishly pursue performance and opt out (or manipulate) management policies when outcomes fail to satisfy their preferences; define *contentiousness* as user demand for shared resources such as memory bandwidth; and define *penalty* as user disutility, such as throughput loss from contention. Cooperative game theory guides us to colocation algorithms that satisfy three system desiderata.

- **Fair Cost Attribution.** More contentious users incur larger penalties from colocation.
- **Satisfied Preferences.** More users colocate with their preferred co-runners.
- **Stable Colocations.** No subset of users benefits by breaking away to share separate subsystem.

2.1.2 Fair Attribution

We argue that a colocation's performance penalties are attributed fairly when more contentious users incur larger penalties. In practice, such fair attribution encourages colocation. Suppose Alice's job is contentious and Bob's is not. If Bob contributes

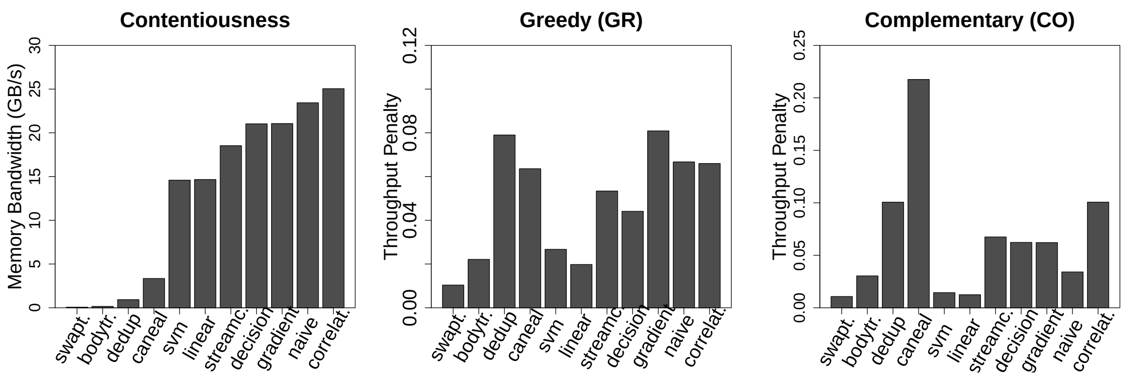


FIGURE 2.1: Unfair colocations show no link between contentiousness and penalties. We colocate 1000 jobs drawn randomly from an application pool. Pairs of jobs share multiprocessor cache and memory bandwidth. Colocation penalties are averaged over those that include a particular job type (e.g., `bodytrack`).

little interference but suffers large performance losses when colocated with Alice, he has little *incentive to share*. Bob would rather form his own private cluster than contribute resources to the shared system. As Bob-like users leave the system, Alice-like users dominate and exacerbate contention.

Figure 2.1 highlights unfairness in existing policies. A greedy policy assigns jobs to servers that perform well given prior assignments. A complementary policy pairs jobs with harmonious demands such as compute and memory intensive jobs. Neither policy links contentiousness to penalty (memory intensity and performance loss, respectively). `Correlation` is the most contentious but penalized no less than `Canneal` and `Dedup` under greedy pairing. `Dedup` is one of the least contentious applications but penalized more than most applications under complementary pairing. These outcomes violate fairness in cost attribution.

Our practical notion of fairness is justified by the notion of Shapley value in cooperative game theory [92]. Shapley determines each agent's fair share of a common outcome based on her contributions. Equation 2.1 shows the Shapley calculation. When applied to colocation, ϕ_i is agent i's fair share of penalty p, which depends on

Coalition (S)	**Penalty (p)**
{A}	0
{B}	0
{C}	0
{A, B}	3
{A, C}	4
{B, C}	5
{A, B, C}	6

Permutation	$\mathbf{M_A}$	$\mathbf{M_B}$	$\mathbf{M_C}$
{A, B, C}	0	3	3
{A, C, B}	0	2	4
{B, A, C}	3	0	3
{B, C, A}	1	0	5
{C, A, B}	4	2	0
{C, B, A}	1	5	0
$\phi_\mathbf{i} = \mathbf{E}[\mathbf{M_i}]$	1.5	2.0	2.5

FIGURE 2.2: Shapley permutes users, calculates their contributions to penalties **M** and expected values ϕ over permutations.

the agents in colocation S.

$$\phi_i(p) = \sum_{S \subset N} \frac{(s-1)!(n-s)!}{n!} \left[p(S) - p(S-i)\right] \tag{2.1}$$

Agent i's marginal contribution to colocation penalties is $p(S) - p(S-i)$. Shapley states that her fair share ϕ_i of penalty p is her marginal contribution to those penalties, averaged over the combinations that colocations could form.

Shapley assigns penalties ϕ that correlate with interference I when system penalty is $p = \sum_{i \in S} \mathrm{I}_i$. System penalty depends on coalition of three users (A,B,C). Shapley permutes users, calculates each user's marginal contribution to penalties **M**, and calculates expected values ϕ over permutations.

We apply Shapley to motivate larger colocation penalties for more contentious users. Consider a simple model of colocation and its contention penalties. Users A, B, and C perform normally when alone but suffer penalties when colocated. Each user contributes interference $\{\mathrm{I_A} = 1,\ \mathrm{I_B} = 2,\ \mathrm{I_C} = 3\}$. Suppose system-wide penalty is the sum of each user's contribution to interference such that $p = \sum \mathrm{I}_i$. Shapley determines users' marginal contributions to penalties, averaged over permutations of users in the coalition – see Equation 2.1.

To understand Shapley, suppose n agents arrive sequentially and $n!$ orderings are equally likely. Agent i arrives after agents in coalition $S - i$ and is the s-th agent in S with probability $(s-1)!(n-s)!/n!$. Agent i's arrival increases coalition penalty by $p(S) - p(S - i)$.

Figure 2.2 enumerates penalties and orderings for our example. Consider ordering {A, C, B}.

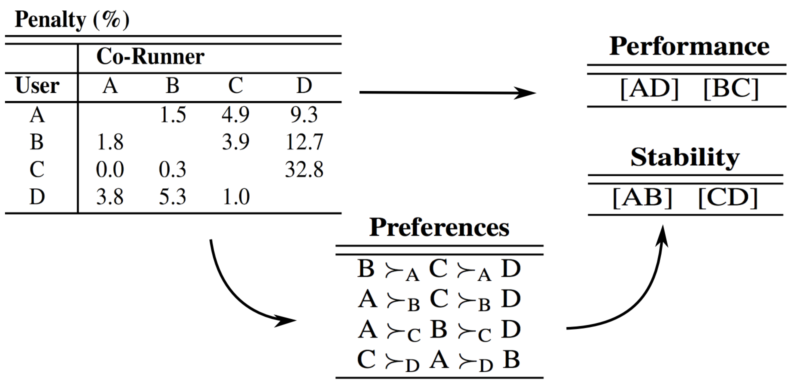
- A's marginal penalty is $\mathbf{M_A} = v(\mathrm{A}) - v(\varnothing) = 0$;
- C's marginal penalty is $\mathbf{M_C} = v(\mathrm{AC}) - v(\mathrm{A}) = 4$;
- B's marginal penalty is $\mathbf{M_B} = v(\mathrm{ABC}) - v(\mathrm{AC}) = 2$.

Each user's Shapley value is her average marginal contribution to penalties across permutations. From Shapley, a fair assignment of penalties is $\phi = \{1.5,\ 2.0,\ 2.5\}$, which correlates with users' contributions to interference $\mathrm{I} = \{1,\ 2,\ 3\}$.

Shapley is an analytical framework, which is not meant for direct application because it unrealistically assumes performance losses can be transferred arbitrarily between colocated agents. Nonetheless, Shapley provides the theoretical foundation for a realistic fairness goal—larger losses for more contentious jobs. In practice, colocation policies pursue this goal by assigning co-runners that dictate performance losses.

2.1.3 Satisfied Preferences and Stability

Stability leads to an equilibrium state which satisfy user preferences and strengthen system integrity. Figure 2.3 illustrates instability from existing policies. Suppose four users share two processors and suffer from contention in the memory subsystem. A colocation policy minimizes system-wide penalties with colocations {AD, BC}. However, these colocations do not satisfy preferences – pairing A with D even though A prefers D least. In addition, they are unstable as A and B prefer each other over their co-runners. If A and B break away to form a separate subsystem to improve their utility, the datacenter fragments and efficiency suffers. In contrast,



Penalty (%)

	Co-Runner			
User	A	B	C	D
A		1.5	4.9	9.3
B	1.8		3.9	12.7
C	0.0	0.3		32.8
D	3.8	5.3	1.0	

FIGURE 2.3: Users' penalties determine preferences for co-runners. Pursuing performance minimizes system penalties. Pursuing stability satisfies user preferences. The four users in this dataset are: (A) `x264`, (B) `fluidanimate`, (C) `decision-tree`, (D) `regression`.

stable colocations {AB, CD} satisfy three of four users' preferences – A, B and D's. Moreover, no pair wants to break away to form their own subsystem.

Figure 2.4 indicates that stability enhances fairness whereas the pursuit of performance does not. When optimizing system-wide performance, user C sees the smallest performance penalty although it is most memory-intensive (1%, 21 GB/s). Users A and B see the largest penalties although they are least contentious (4-9%, 4-5 GB/s). In comparison, stability-centric policies more closely align penalties with memory intensity. The penalty for the most contentious user rises while those for less contentious users fall. Stability, although imperfect, furthers the fair attribution of costs in shared systems.

2.2 The Colocation Game

We present a game-theoretic framework that colocates software on shared hardware in a multi-user setting. Our framework is an alternative to heuristics that myopically maximize performance. The colocation game balances the pursuit of performance with the provision of fairness, which encourages strategic users to share hardware.

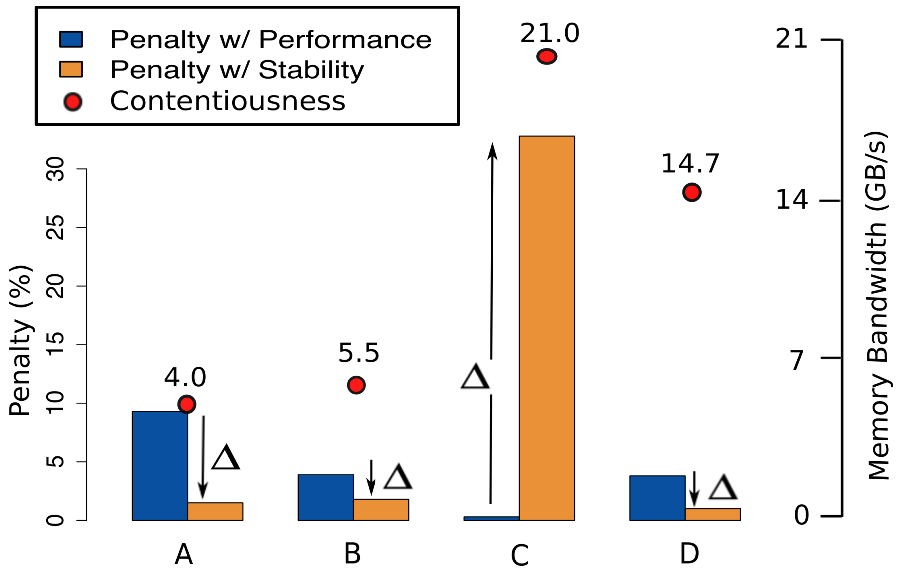


FIGURE 2.4: Stability enhances fairness. Bars show user penalty based on throughput loss under performance- and stability-centric colocation policies. Dots show user contentiousness, based on bandwidth demand when alone. Data for four users: (A) `x264`, (B) `fluidanimate`, (C) `decision-tree`, (D) `regression`.

2.2.1 System Setting

We consider a shared cluster with homogeneous processors, each with multiple cores, that serve batch and offline computation. The colocation game batches and assigns arriving jobs to available processors periodically. The length of the scheduling period is comparable to job completion times (i.e., minutes rather than seconds or milliseconds). If the system is heavily loaded, jobs queue for scheduling.

Figure 2.5 illustrates an architecture for the colocation game, which defines abstraction layers – agents and coordinator – between users and machines. Agents act on users' behalf within the game, shielding users from complex management mechanisms. The coordinator communicates system information to agents and implements management mechanisms.

Agents play three roles when interfacing with the coordinator. First, agents query the coordinator's profiler to obtain job performance under varied colocations. Second,

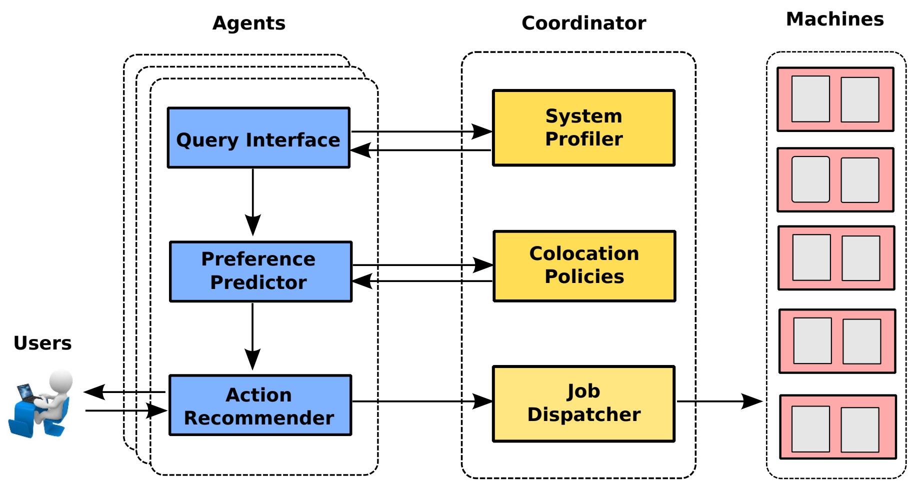


FIGURE 2.5: Agents act on users' behalf, playing the colocation game and interfacing with the system coordinator. The agents and the coordinator shield hardware complexity from human users.

they use profiles to predict preferences for co-runners and influence the coordinator's colocation assignments. Third, agents assess assigned colocations and recommend strategic actions to users. An agent recommends participation when assignments satisfy preferences. Otherwise, an agent recommends better colocations with others.

2.2.2 Game Formulation

We formulate colocation as a cooperative game in which users form coalitions to share hardware and divide penalties from resource contetion. We define the game's components, introduce actions, and present solution concepts.

Agents, Disutility, and Preferences. An agent represents a user and her job. In a given epoch, the colocation game assigns $2N$ agents to N chip multiprocessors. Colocated agents comprise a coalition who contribute to shared contention and performance penalties. Each agent defines disutility $d \in [0,1]$.

$$d \;=\; 1 - \frac{\text{Throughput}_{\text{ colocation}}}{\text{Throughput}_{\text{ stand-alone}}}$$

Disutility quantifies a colocation's performance penalty. For example, $d = 0.3$

when a job's colocated performance is $0.7\times$ that of its stand-alone performance, all else being equal (e.g., allocation of processor cores). Disutility dictates an agent's preferences for co-runners. Let $\succ_i$ denote agent i's preferences. If i's disutility with x is lower than its disutility with y, then $x \succ_i y$. In other words, i performs better with x than with y.

Strategic Action. The datacenter operator would like all agents to share one monolithically managed cluster to enhance efficiency. However, subsets of agents could determine that a colocation policy provides better individual outcomes when applied to separately managed clusters. Agents would then create subsystems shared by mutually preferred co-runners. Breaking away is the act of finding a subset of agents who form new coalitions on separately shared subsystems to improve their performance.

Blocking Coalitions and Equilibria. Agents who break away to pursue better outcomes together comprise a blocking coalition. Let C denote a datacenter's colocations and $C(i)$ denote i's co-runner, assuming two users share a chip multiprocessor. Agents i and j are blocking if they prefer each other over their co-runners: $j \succ_i C(i)$ and $i \succ_j C(j)$. Colocations with fewer blocking pairs are more stable.

Stability is a system outcome that minimizes the number of blocking pairs, producing equilibria in which all agents participate in the shared system. In equilibrium, no subset of agents can better satisfy preferences and improve performance by deviating from assigned colocations. In contrast, neglecting preferences produces blocking pairs and harms stability.

2.2.3 Game Solutions

Stable matching is a natural fit for colocation. A matching process builds pairwise coalitions based on mutual consent from independent, strategic agents. Matches are stable when no pair of agents prefers each other over their existing partners. We draw

Algorithm 1 Stable Marriage for Colocation Game

```
1: sets M, W ← 2N tasks such that |M| = |W| = N
2: lists P[i] ← ordered preferences ∀i ∈ M, W
3: single(i) ← True ∀i ∈ M, W
4: while ∃single(m) ∈ M, P[m] ≠ ∅ do
5:     w ← P[m]
6:     if single(w) then
7:         pair(m, w)
8:     if (m', w) paired, but m ≻_w m' then
9:         pair(m, w)
10:        single(m') ← True
11:    P[m] ← P[m].next
```

inspiration from stable algorithms for marriage [40] and roommate assignment [51], adapting them to the colocation game.

Stable Marriages. The stable marriage algorithm solves the colocation problem with two sets of agents. Agents in one set propose colocations while those in the other accept or reject them. Agents act strategically to pursue their preferred co-runners.

Algorithm 1 sketches the procedure for finding stable marriages between two sets of jobs, which are labeled M and W. Job m proposes to w according to its ordered preferences. Job w accepts when it prefers m over its current co-runner m'. If w rejects, m proposes to its next preferred co-runner. The procedure iterates until all jobs are matched. In each round, all jobs in M propose to their top-ranked co-runners simultaneously. Each job in W accepts its best proposal and rejects the rest in parallel. Those in M that are not accepted proceed to the next round. The procedure continues until all jobs are matched.

The procedure provides stable colocations in which no two agents from opposite sets can break away and improve their utility [35]. Every job in M has one successful proposal because a job in M that had all prior proposals rejected is accepted by the least desirable job in W. Stability arises from accepted proposals. Suppose m prefers w' over its co-runner w. Because m and w' are not colocated, m must have proposed to w' only to have been rejected because w' preferred m'. Matches are stable because $m' \succ_{w'} m$ even though $w' \succ_m w$.

Preferences
$m_1 : c_1 \succ c_2 \succ c_3$
$m_2 : c_3 \succ c_1 \succ c_2$
$m_3 : c_1 \succ c_2 \succ c_3$
$c_1 : m_2 \succ m_3 \succ m_1$
$c_2 : m_3 \succ m_1 \succ m_2$
$c_3 : m_2 \succ m_1 \succ m_3$

Round	Propose	Accept	Reject
1	$m_1 \rightarrow c_1$	$c_1 - m_3$	m_1
	$m_2 \rightarrow c_3$	$c_3 - m_2$	
	$m_3 \rightarrow c_1$	$c_2 -$	
2	$m_1 \rightarrow c_2$	$c_2 - m_1$	

FIGURE 2.6: Stable marriage with compute- and memory-intensive jobs.

Adapting Partitions and Proposals. Stable marriage matches jobs from two disjoint sets, requiring a job partitioning strategy. Some strategies arise from the system. High- and low-priority jobs should be partitioned, as should compute- and memory-intensive jobs. When domain expertise indicates jobs within a set should not colocate with each other, marriage is a solution that precludes intra-set matches.

The algorithm can also partition jobs randomly. In large systems with diverse jobs, random partitions uniformly distribute jobs of all types across two sets. Stable marriages are more likely when each set holds diverse jobs, not just memory-intensive ones. Diverse preferences produce diverse proposals and reduce the likelihood of common, desirable co-runners. Random partitions are as effective as sophisticated ones for satisfying preferences.

We implement two partitioning mechanisms — partition based on applications' memory intensity and partition randomly. Partitioning by memory intensity reflects the source of hardware contention and tends to favor performance. Partitioning randomly neglects inherent job characteristics and tends to favor fairness.

Agents that propose perform nearly optimally and better than those that receive proposals [52]. Proposers choose co-runners in order of their preferences where as those that receive proposals have no influence on their suitors. Agents accept or reject without knowing which job might propose next. In practice, we find that proposers' advantages are small, especially for randomly partitioned jobs.

Stable Marriage Example. Figure 2.6 presents an example of stable marriage.

First, the system partitions memory- and compute-intensive jobs (m and c), based on memory bandwidth demands. Second, agents profile and predict preferences, ranking candidate co-runners in the opposite set. Finally, jobs in set m propose to those in set c. Specifically, m_1 and m_3 both propose to c_1. Based on its preferences, c_1 accepts m_3 and rejects m_1. Simultaneously, m_2 proposes to c_3, which accepts as it lacks a better proposal. Rejected, m_1 proposes to c_2 in the next round. Lacking a proposal, c_2 accepts and the algorithm terminates with colocation $\{m_1c_2,\ m_2c_3,\ m_3c_1\}$.

Stable Roommates. Roommate assignment provides a natural alternative to marriage when an agent may match with any other. Irving provides a generalized matching algorithm [51]. First, each agent proposes sequentially to preferred roommates while simultaneously receiving proposals from others. An agent rejects a proposal if she already holds a better one and accepts otherwise. If any agents are rejected by everyone, the algorithm terminates and states that no perfectly stable solution exists. If all agents hold successful proposals, each agent reduces her preference list by deleting roommates that are less desirable than proposals they hold. The algorithm further reduces preference lists by eliminating preference cycles (e.g., $B \succ_A C$, $C \succ_B A$, $A \succ_C B$). The algorithm terminates when no cycle exists and produces stable roommate assignments.

Adapting Stable Roommate. Stable roommate assignment does not always produce a solution. We extend the algorithm with heuristics when no stable solution exists. When Irving's algorithm terminates with no solution, we greedily pair unmatched agents to minimize their pair disutilities. In practice, stable roommate assignments rarely exist for large agent populations. For such settings, our adapted algorithm significantly reduces the number of blocking pairs.

Limitations. In theory, stable-matching problems with arbitrary group size can not be solved in polynomial time. Only in the instance of a pair-wise stable roommate problem, does a polynomial algorithm exist [51]. However, if no stable matching exists, minimizing the number of blocking pairs is NP-hard and hard to approximate [6]. One approximation relaxes stability, defining an α-stable system in which agents form blocking pairs only when gains exceed a factor of α [12]. α-stability is particularly relevant when agents set thresholds that reflect the costs of breaking away. For example, colocated jobs deploy separately managed subsystems only to avoid large performance penalties.

In practice, finding a stable matching or $\alpha-$stable matching for an arbitrary group size is hard and may not be necessary. When colocating more than two data intensive applications on the same processor, the resource competition caused in the memory subsystem (share last level cache and memory bandwidth) is large and causes high performance penalties. This system scenario happens rarely.

In sum, stable matching solves the colocation game efficiently. The solution satisfies preferences and preempts strategic behavior. In theory, marriage and roommate algorithms find pairwise matches in polynomial time. In practice, overheads are modest in our implementation.

2.3 Cooper Colocation Framework Design

We design and implement `Cooper`, a cooperative game that provides the management desiderata, fair colocations for strategic users, satisfies users' preferences and stability. It enhances shared user fairness, safeguard the integrity and efficiency of the shared system.

Figure 2.7 illustrates `Cooper`'s architecture and components. Decentralized agents act on behalf of users to pursue preferred colocations. Each agent instantiates three modules. The query interface requests profiles for sparsely observed colocations. The preference predictor estimates performance for unobserved colocations. The action recommender assesses assigned co-runners and suggests user action.

To support agents, `Cooper` implements a centralized coordinator with three mod-

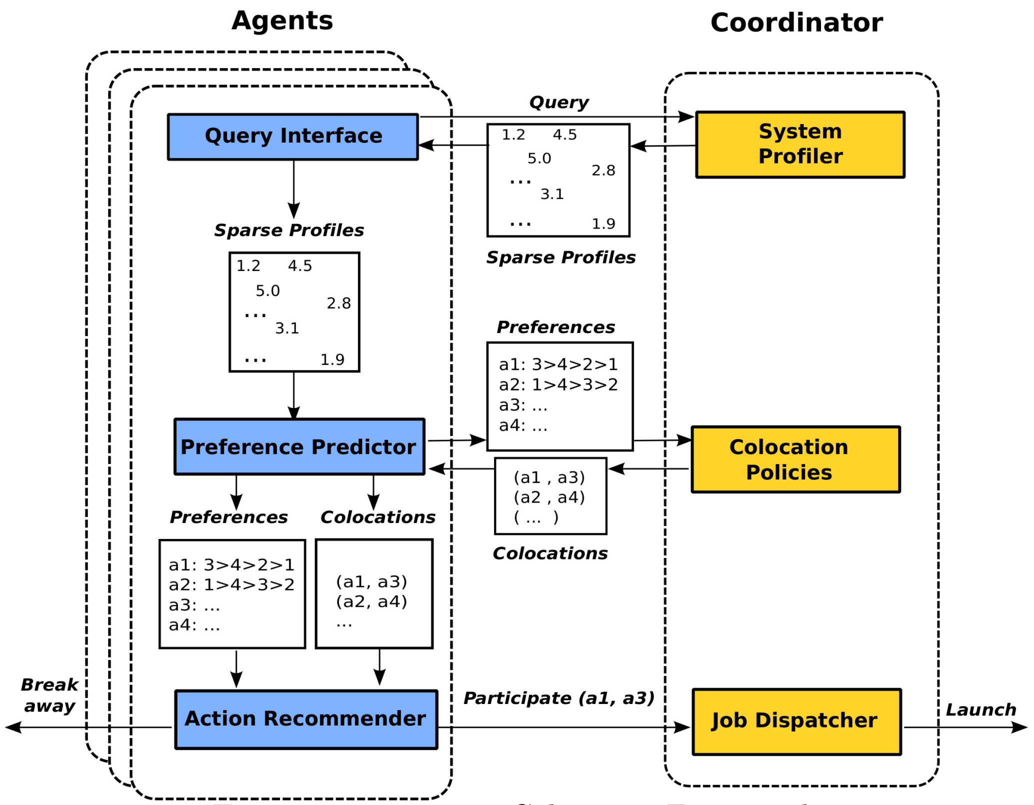


FIGURE 2.7: `Cooper` Colocation Framework

ules. The system profiler responds to queries with a database of performance measurements. Colocation policies assign co-runners based on agents' preferences. The job dispatcher assigns computation to machines when agents choose to participate.

`Cooper`'s design emphasizes intelligent agents that separate strategic users from the shared system. From the user's perspective, the system delivers fairness and stability to encourage participation. Users rely on agents to assess colocations and recommend strategic action. From the system's perspective, agents pursue preferred colocations independently.

2.3.1 Preference Predictor

The predictor receives performance profiles and estimates preferences for co-runners. It uses sparsely profiled colocations to infer a preference list that ranks co-runners by the agent's expected performance. The game's matching algorithms use preferences to find stable colocations.

In principle, users could report preferences directly to the system coordinator; however, they are poorly equipped to assess preferences for each co-runner. Because self-reported preferences can be burdensome, inaccurate, and non-truthful, `Cooper` relies on agents' predictors.

Collaborative Filtering. Agents employ light-weight predictors to estimate preferences. Determining preferences for each co-runner via direct measurement is intractable. Fortunately, predicting agents' preferences from sparse performance profiles is analogous to predicting consumers' preferences from sparse product ratings. Predictors treat jobs as consumers, co-runners as products, and profiles as ratings.

Collaborative filtering trains predictors, observing that consumers who rate many items similarly share preferences for other items. `Cooper` implements item-based collaborative filtering, predicting that a co-runner affects similar agents similarly. When a co-runner degrades one task's performance, it will similarly degrade another's. `Cooper` implements IBCF because it emphasizes similarity in contention's sources and effects. If $x_1 \succ_{y_1} x_2$ and x_3 is similar to x_2, IBCF infers that $x_1 \succ_{y_1} x_3$.

Implementation. We preferences using an R library – `recommenderlab` [74]. For n agents, a sparse $n \times n$ matrix $M[x, y]$ reports x's performance with co-runner y. In each iteration, the recommender predicts the unknown ratings in the matrix while minimizing error for known values. Iterations terminate when all matrix elements are filled. In practice, this process requires one to three iterations and completes within 100ms for 1000 agents.

Sparsity affects accuracy. `Cooper` trains the recommender with 25% sparsity. With 20 unique jobs, Cooper uses 100 $(20 \times 20 \times 0.25)$ sampled colocations to predict the dense matrix. Our experiments indicate that error is unacceptably high with 20% of profiles sampled, falls quickly with 25%, and falls slowly beyond 30%.

Predictor Design Discussion. This part discusses alternative approaches to implement the preference predictor. Preferences measure relative performance, permitting more accurate inference from profiles. Relative measures can be transitive. When $B \succ_A C$ and $C \succ_A D$, then $B \succ_A D$. Moreover, relative measures can be predicted even when platforms change. If $B \succ_A C$ on one machine, this preference likely holds on another.

Classifying n jobs into $t < n$ types shrinks matrix dimensions and reduces training overheads [27] at the expense of less-precise preference predictions because individual jobs cluster into generic types. We apply collaborative filtering directly, using profiles for only a fraction of possible colocations to control overheads.

Content-based filtering, an alternative approach, identifies attributes that affect consumer ratings and recommends items that rank highly for those attributes. Agents may prefer co-runners with small cache footprints or bandwidth demands. Unfortunately architects must identify the most relevant attributes, which can be difficult and complicated by software phases.

Other heuristics might predict colocation preferences. Conventional wisdom states that memory-intensive jobs prefer compute-intensive co-runners, or jobs prefer co-runners with smaller working sets. In practice, we find that rules of thumb based on high-level attributes and complementary resource demands do not deliver game-theoretic desiderata.

2.3.2 Action Recommender

The coordinator receives predicted preferences from agents and assigns co-runners. Agents assess assignments and recommend strategic action – participate or break away – for their users. If breaking away is recommended, the agent identifies separately managed colocations (i.e., blocking pairs) and their expected performance advantages.

Dissatisfied agents seek opportunities to break away. The agent assesses its assigned co-runner by exchanging messages with others. It sends messages to agents ahead of its assigned co-runner in its preference list. Conversely, it receives such messages from other agents. Suppose agent X has preferences $A \succ_X B \succ_X D \succ_X E$ and is assigned co-runner D. X sends messages to A and B. If X receives messages from A or B, it knows $X \succ_A C(A)$ or $X \succ_B C(B)$, meaning that A and B both prefer X than their assigned co-runners. Agents A and B would recommend breaking away and forming a separate system.

Implementation. We implement the action recommender as a Java application within each agent. Agents communicate via network and files. Agents return, to human users, lists of blocking pairs with suggestions to participate or break away. In our implementation, agents participate and invoke the job dispatcher by default. We then assess fairness by counting blocking pairs created by a colocation policy. Users in a blocking pair would break away given agents' suggestions and her performance goals.

2.3.3 Colocation Policies

The coordinator receives preferences and returns colocations. We implement matching algorithms to solve the colocation game. We compare game-theoretic solutions to two baselines that reflect conventional wisdom.

- **Stable Marriage Partition (SMP)** partitions tasks by resource demands and pairs tasks with stable marriage. Resource-intensive set proposes.

- **Stable Marriage Random (SMR)** partitions tasks randomly and pairs tasks with stable marriage. Randomly selected set proposes.

- **Stable Roommate (SR)** pairs tasks with stable roommates. When no stable solution exists, SR employs GR to pair tasks rejected by all others.

- **Greedy (GR)** assigns each task, sequentially, to the processor that minimizes contention given prior assignments.

- **Complementary (CO)** partitions tasks by resource demands and pairs tasks with complementary demands.

Threshold schemes colocate jobs when penalties are less than 10%, for example, and add a new machine otherwise [70]. When no machine is held in reserve and ready to supply capacity, GR performs at least as well as a threshold. GR minimizes penalties whereas a threshold permits penalties up to specified tolerance.

Implementation. We implement colocation algorithms in Java and output co-runner assignments to files, which are sent to agents. For n agents, stable matching employs $O(n^2)$ algorithms and the complementary mechanism employs an $O(n)$ heuristic. When necessary, jobs are sorted and partitioned by resource demands with $O(n\log n)$ algorithms. Measured overheads are modest. To colocate 1000 agents, stable matching requires 1 to 5 seconds. In comparison, job completion times range from 10 to 15 minutes for Spark and from 2 to 5 minutes for PARSEC.

2.3.4 Other Components

System Profiler. Modern systems can profile any job on any machine. Google samples servers, profiles continuously, and builds databases that support SQL-like queries [90]. Queries with job IDs, machine IDs, and timestamps retrieve performance for varied colocations. We construct a database for 20 open-source jobs.

Offline, the profiler measures performance for standalone jobs and sampled colocations. We measure Spark task throughput, modifying the engine (v1.6.0) to log task, stage, and job completion. We measure PARSEC runtimes with `perf stat`. For microarchitectural profiles (e.g., memory bandwidth), we read MSR registers once per second with Intel's Performance Counter Monitor 2.8.

Online, the profiler responds to queries with a sparse matrix of performance penalties for sampled co-runners. Sampling is required for tractability, especially at datacenter scale. Preference predictors accommodate sparsity, requiring profiles for only a small fraction of possible colocations.

Job Dispatcher. The job dispatcher sends computation to machines. After the coordinator assign co-runners and agents choose to participate, the dispatcher sends jobs' binaries and data to available machines. Each machine runs a daemon that checks periodically for work. This simple implementation extends naturally to managers such as Spark, Yarn, or Mesos [44, 102, 118].

2.4 Experimental Methodology

Workloads. Table 3.2 summarizes evaluation benchmarks from Spark [117] and PARSEC 2.0 [17], which are representative of batch computation and data analytics. Methods for multiprogrammed benchmarking vary [54]. We repeat the shorter workload until the longer one completes. We do not consider latency-sensitive applications, such as search, as their stringent targets for service quality often preclude colocation [27, 64].

Figure 2.8 presents performance losses. Entry $[x, y]$ shows x's penalty when colocated with y. For example, penalty is 0.3 when colocated performance is $0.7\times$ standalone performance.

Agent Populations. We evaluate the colocation game with large, diverse agent populations. We evaluate 1000 agents, sampling jobs uniformly at random with replacement from Table 3.2. After agents receive and assess co-runners, the coordinator dispatches jobs. Jobs dispatch in batches when the system has fewer multiprocessors than colocated pairs.

Servers. We use a cluster with five nodes, each with two Intel Xeon E5-2697 v2 chip-multiprocessors (CMPs). Each CMP has 12 cores and 24 threads, running

ID. Name	Application	Dataset	GBps
	Apache Spark		
1. Correlation	Statistics	kdda'10 [97]	25.05
2. DecisionTree	Classifier	kdda'10	21.03
3. Fpgrowth	Mining	wdc'12 [5]	10.06
4. Gradient	Classifier	kdda'10	21.06
5. Kmeans	Clustering	uscensus [4]	0.32
6. Regression	Classifier	kdda'10	14.66
7. Movie	Recommender	movielens [3]	5.69
8. Bayesian	Classifier	kdda'10	23.44
9. SVM	Classifier	kdda'10	14.59
	PARSEC		
10. Blackscholes	Finance	native	0.99
11. Bodytrack	Vision	native	0.15
12. Canneal	Engineering	native	3.34
13. Dedup	Storage	native	0.93
14. Facesim	Animation	native	1.80
15. Fluidanimate	Animation	native	5.52
16. Raytrace	Visualization	native	0.57
17. Stream	Data Mining	native	18.53
18. Swaptions	Finance	native	0.07
19. Vips	Media	native	0.05
20. X264	Media	native	4.00

Table 2.1: Application configurations, datasets, and memory intensity.

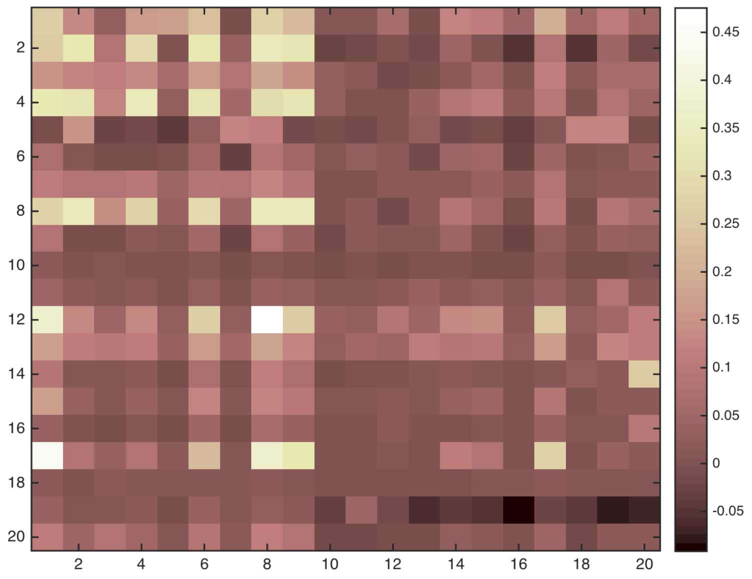


FIGURE 2.8: Disutility from pairwise job colocations. See Table 3.2 for job IDs.

at 2.7GHz and sharing 128GB of main memory. Colocated jobs divide the CMP's threads equally, sharing cache capacity and memory bandwidth. The server configuration focuses on memory contention. Nodes have solid-state drives and 1Gbps Ethernet, precluding I/O and network contention.

2.5 Evaluation

We evaluate system desiderata: (i) fair attribution such that more contentious users incur larger penalties, (ii) satisfied preferences such that more users colocate with preferred co-runners, and (iii) stable colocations such that fewer users break away. Moreover, we show that `Cooper` performs nearly as well as heuristics that minimize contention.

2.5.1 Fairness and Desiderata

Fair Attribution of Costs. Figure 2.10 and 2.9 evaluate fairness by showing the relationship (or lack thereof) between jobs' resource demands and colocation penalties. The x-axis presents jobs ordered by increasing memory intensity. The y-axis presents each job's throughput loss, averaged over its varied colocations when randomly sampled jobs share the system. When bars extend up and right, penalty is proportional to contentiousness and costs are fair.

Conventional policies neglect fairness. GR is unfair as `dedup` demands among the least from shared memory but is penalized most. `Bodytrack` contributes much less to contention than `svm` but suffers the same penalty. Similarly, CO shows no link between application contentiousness and colocation penalties.

Stable policies can enhance fairness, but mixing them with conventional wisdom does not work. SMP builds atop CO, partitioning jobs into two sets based on memory intensity before invoking stable marriage. However, SMP ignores the fact that jobs in one set could prefer each other over jobs in the opposite set. Restricting permissible

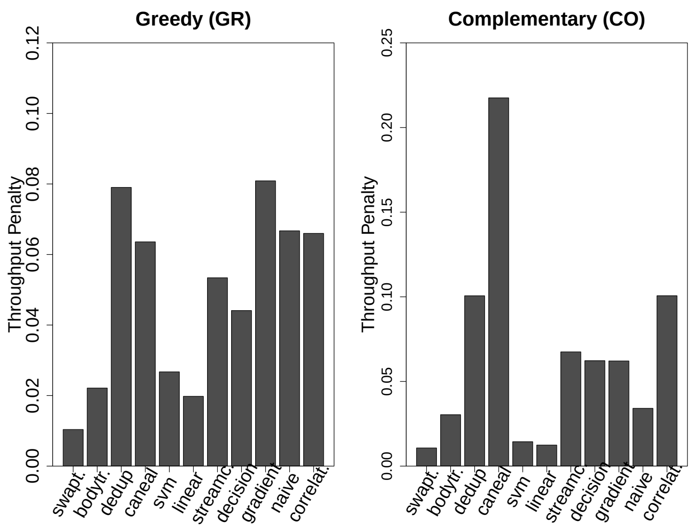


FIGURE 2.9: (a) and (b) show contention-induced performance losses from conventional colocation policies. Jobs are ordered by increasing contentiousness on x-axis.

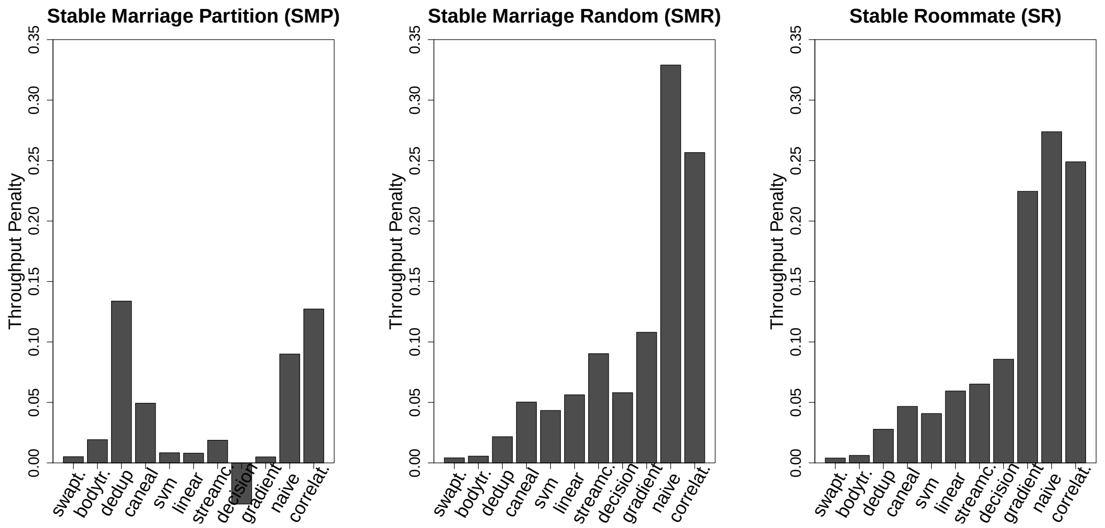


FIGURE 2.10: (a) (b) and (c) show performance losses from stable colocation policies. Jobs are ordered by increasing contentiousness on x-axis.

matches overrides preferences and induces unfairness.[1]

Stable matching improves fairness in less structured game formulations. SMR partitions jobs randomly such that, with some probability, a job might colocate with any other to satisfy preferences. SR permits unrestricted matches. Both SMR and SR produce colocations in which jobs' performance penalties increase with their contentiousness.

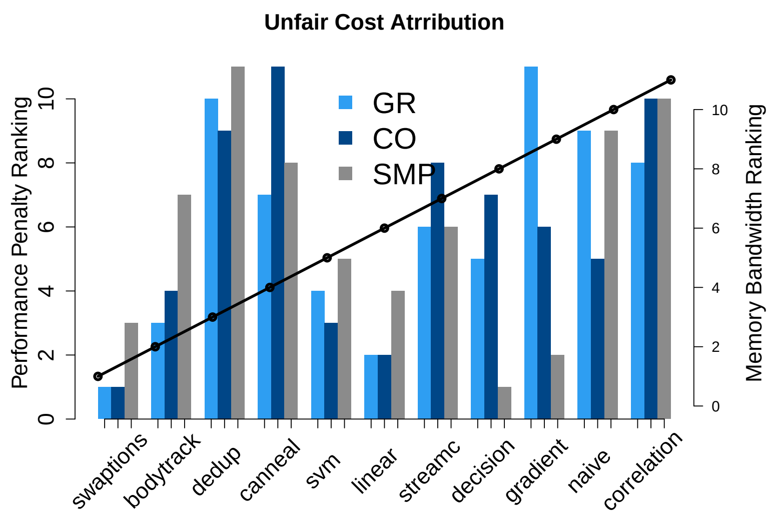
Figure 2.11 illustrates relative fairness, ranking each job's penalty and bandwidth demands. For example, `Swaptions` ranks first with the smallest performance penalties and bandwidth demands while `correlation` ranks 10th in penalties and 11th in demands. Bars present ranked penalties and the line presents ranked demands, which is linear because jobs are ordered by contentiousness on the x-axis.

Bars that track the line illustrate equal treatment of equals and unequal treatment of unequals in proportion to their differences. GR, CO, and SMP are unfair as ranked penalties are unrelated to ranked demands. In contrast, SMR and SR are fair as more demanding jobs experience larger penalties.

Satisfied Preferences. Figure 2.12 shows how stable colocations satisfy more users' preferences. Bars show the number of agents with improved, degraded, or unchanged performance when switching from conventional colocations (GR, CO) to stable ones (S*). For example, choosing stable roommate over greedy colocation improves performance for more than half of the agents – see SR/GR.

A large majority of agents performs at least as well, if not better, with colocations that reflect preferences. Among stable policies, SR performs best as each agent proposes to all others according to its preferences. SMR and SMP perform slightly worse as partitions restrict proposals and satisfy fewer preferences. The minority who suffer larger penalties are those held responsible for their larger contributions

[1] Tasks occasionally perform better colocated than alone due to variance across system measurements.



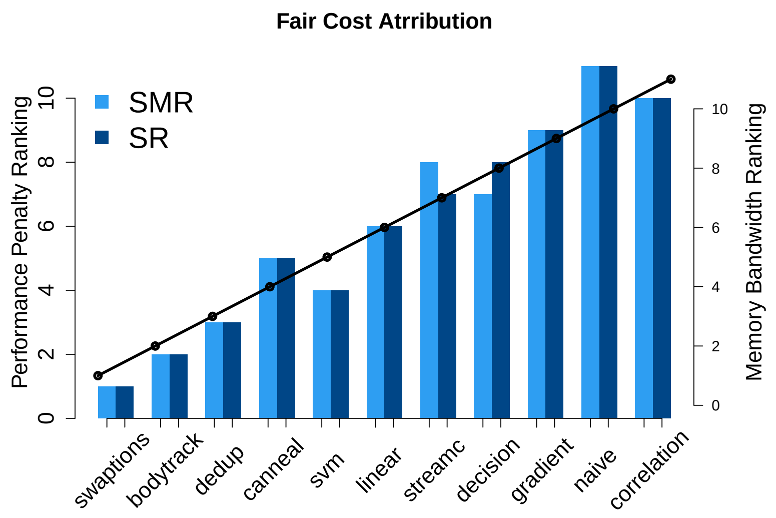


Figure 2.11: Correlation between ranked performance penalties (bars) and bandwidth demands (line). When the bars track the line, colocations are fair. See Figures 2.10 for absolute measures of performance and bandwidth.

to contention, a fair outcome.

Stable Colocations. Figure 2.13 counts agents that recommend breaking away from assigned colocations for new, mutually beneficial ones. Boxplots present the distribution of these counts for 50 populations of 1000 sampled jobs. Parameter α is the minimum performance benefit for which an agent breaks away. Increasing α reduces the number of blocking pairs and improves stability. Assuming an agent's original

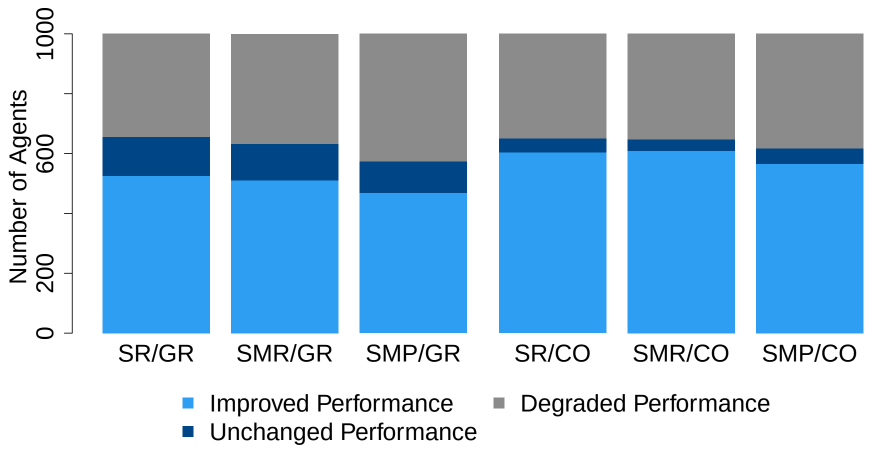


FIGURE 2.12: Performance impact when adopting cooperative game (S*) instead of performance-centric policies (GR, CO). Data is averaged over 10 populations, each with 1000 randomly sampled jobs.

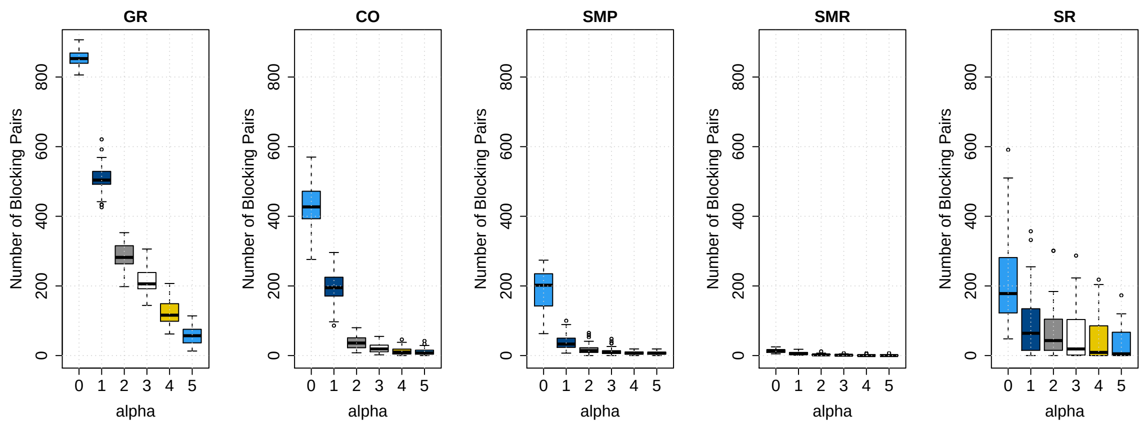


FIGURE 2.13: Stability analysis, which measures the number of blocking pairs (y-axis) for varied policies and α (x-axis), the minimum benefit for which an agent breaks away. When α=2%, agents break away for new colocations that improve both agents' performance by 2%. Here, we show data distributions and boxplots for 50 populations, each with 1000 randomly sampled jobs.

performance penalty is 0.2, if $\alpha\% = 0.5$, it means that if this agent's performance penalty is reduced to 0.15 with the new colocation, she will prefer to deviate.

GR colocations are less stable, ignoring preferences in pursuit of performance and producing dissatisfied agents. In contrast, CO produces fewer blocking pairs, especially when agents break away only for large gains (e.g., α=5%). By pairing complementary jobs, CO bounds performance penalties and avoids instability. But it delivers neither fair attribution nor satisfied preferences.

SMR colocations are most stable. Its random partitions reduce the likelihood that an agent prefers but cannot match with co-runners in its own set, which is a major cause of blocking pairs. SMR distributes contentious tasks across two sets, reducing agents' risks of poor matches. Note that we count blocking pairs wherever they arise. If the population is partitioned, agents in a blocking pair could belong to the same or opposite set.

SMP and SR are less stable because they force some agents into undesirable matches. SMP places contentious agents into the same set. Less contentious agents cannot match with each other and must match with opposite agents, which creates blocking pairs. Although SR finds stable solutions if they exist, they rarely do and heuristics that match agents rejected by all others create blocking pairs.

Summary. Stable Marriage Random most effectively delivers system desiderata – fair attribution, satisfied preferences, stable colocations. Fortunately, SMR is also the easiest to implement. It always produces a solution and randomly partitioning agents needs no extra profiling.

2.5.2 Performance and Sensitivity

Figure 2.14 and 2.15 present performance penalties and assesses sensitivity to workload mix. We vary the probability density used to sample jobs that comprise an agent population. Thus far, we have used the Uniform density, in which every job is

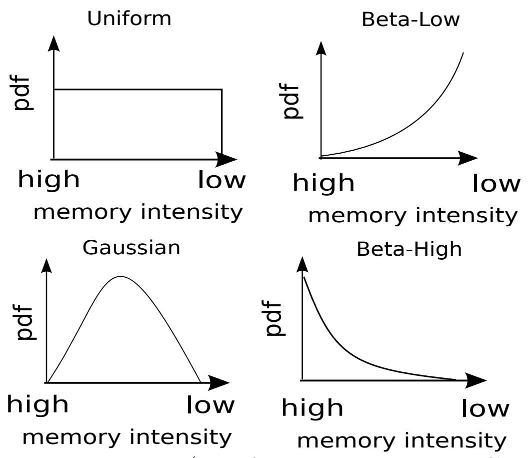


FIGURE 2.14: Workload mixes (Uniform, Beta-Low, Gaussian, Beta-High).

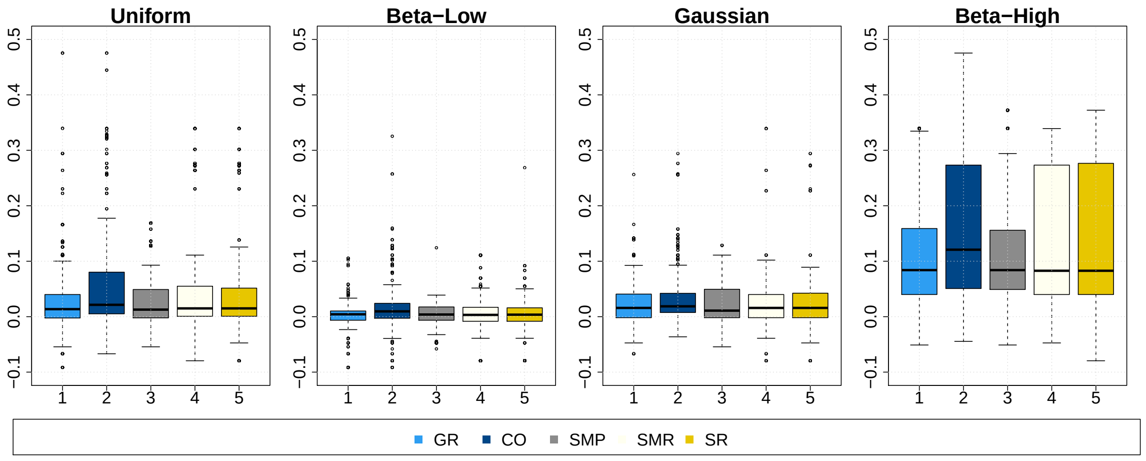


FIGURE 2.15: Performance penalties measured for varied colocation polices (GR, CO, SMP, SMR, SR) for various workload mixes presented in Figure 2.14. Each box shows the performance penalty distribution of 1000 colocated tasks.

represented equally. The Beta density represents populations skewed toward more or less memory intensive jobs. The Gaussian density represents populations of moderate jobs.

In theory, the performance gap between optimal and stable colocations is unbounded [52]. In practice, stable policies (S*) perform as well, if not better, than conventional ones (GR, CO). Penalties are larger when the Beta density skews populations toward memory-intensive jobs. SMP performs best, avoiding large penalties by partitioning jobs such that contentious jobs cannot match with each other. The Beta-High density with many contentious jobs is a challenging scenario, requiring effective policies and more resources for service quality.

Some systems specify penalty thresholds, accepting colocations if performance degrades less than some tolerance (e.g., 10%). By this measure, stable policies (S*) perform comparably with GR and better than CO. The upper whisker, which is 3$\times$ the inter-quartile range away from the third quartile, is within tolerances. Service quality from fair policies, which may sacrifice performance for contentious jobs, is comparable to that from conventional policies.

Summary. The colocation game delivers desiderata with little effect on performance. Stable and conventional policies perform similarly for varied system scenarios. A pessimistic scenario with many contentious tasks reveals a particularly advantageous policy – stable marriage with partitions.

2.5.3 Preference Prediction

Figure 2.16 evaluates collaborative filtering and the accuracy of its predicted preferences. The rank coefficient τ compares a predicted list against the true list, counting inconsistencies.

$$\tau = 1 - \left[\sum_{a \in A} \sum_{i,j \in C_a} K_{ij}\right] \times \left[n\binom{n}{2}\right]^{-1} \tag{2.2}$$

The double summation counts incorrect predictions across agents $a \in A$ and potential matches $i, j \in C_a$ for each agent. $K_{ij} = 1$ when an agent's preference for i relative to j differs across true and predicted preferences, and $K_{ij} = 0$ otherwise. The number of incorrect predictions is divided by the number of pairwise preferences and subtracted from one to calculate the fraction of correct predictions.

Figure 2.16 indicates the accuracy of collaborative filtering improves with more data, starting at 83% with 25% of colocations profiled and rising to 95% with 75%

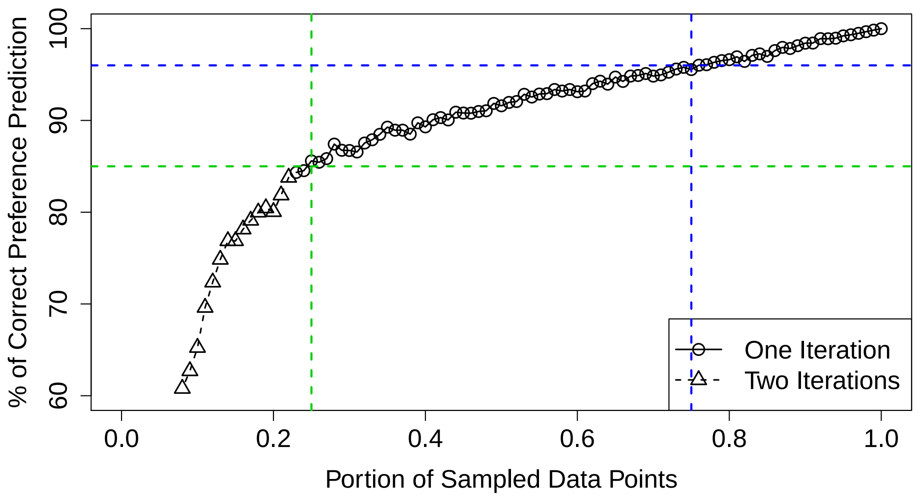


FIGURE 2.16: Prediction accuracy, which evaluates the percentage of correctly predicted preferences (Equation 2.2). x-axis shows various sample ratios.

profiled. With such accuracy, our stable policies deliver the same desiderata whether using oracular knowledge or collaborative filtering.

2.5.4 Scalability

Figure 2.17 evaluates fairness as the number of agents increases. For SMR, the correlation between a job's performance penalty and bandwidth demand strengthens with more agents. Smaller populations exhibit less diversity across jobs, hindering the search for matches that satisfy preferences. Larger populations increase the likelihood that an agent finds a satisfactory co-runner. Standard deviations shrink with population size, reducing the risk of unfairness. `Cooper` is more effective for larger systems with hundreds of multiprocessors.

2.6 Related Work

Fair Resource Management. Computer architects have explored hardware mechanisms for fair sharing in chip multiprocessors, especially when partitioning caches or scheduling memory accesses [31, 77, 79, 86]. We develop a system-level colocation

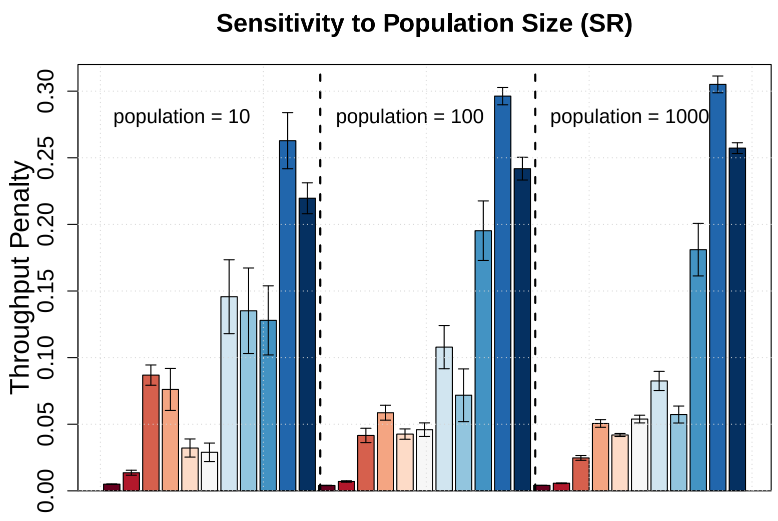


FIGURE 2.17: Scalability analysis and SMR fairness as the number of agents increases. Link between contentiousness and penalty is weak in small systems. In larger systems, more contentious jobs have larger penalties.

framework to management memory subsystem contention.

Many studies focus on game-theoretic desiderata when allocating resources to strategic users whereas we focus on such desiderata for colocation, a novel objective. Ghodsi et al. propose Dominant Resource Fairness to allocate cores and memory [36]. Zahedi et al., proposes Resource Elasticity Fairness using Cobb-Douglas utility function for cache and memory bandwidth allocation [119]. DRF and REF guarantee sharing incentives, Pareto efficiency, envy-freeness, and strategy-proofness. Grandl et al. propose Tetris [38], a multi-resource colocation mechanism that assigns tasks to machines according to resource demands. These studies assume hardware isolation and neglect interference. We pursue game-theoretic desiderata for contentious colocations on bare metal.

Colocation and Scheduling. Prior studies focus on modeling contention and anticipating performance penalties, but neglect preferences and fairness during colocation. Mars et al. predict contention in shared memory systems [70, 113]. Delimitrou et al. models interference and machine heterogeneity with recommenders

[27]. Multiple studies schedule complementary workloads on chip multiprocessors [33, 55, 62, 112, 122].

The discussion of related work should separate colocation profiling and policy. Prior studies provide sophisticated profilers to predict contention and drive simple, greedy policies. In contrast, `Cooper` is a sophisticated policy balances performance and fairness.

On profiling, Bubble-Up/Flux predict contention between colocated jobs, Bubble-Up for two co-runners and Bubble-Flux for more. In contrast, `Cooper` uses recommendation system to predict colocation preferences. On policy, Bubble-up/flux assign jobs to machines when penalties $<10\%$. When a job cannot colocate given this tolerance, it adds a machine. In contrast, `Cooper` colocates applications with limited machines. When extra machines are unavailable, our greedy baseline performs at least as well as the threshold policy.

Cooperative Games and Systems. In mobile systems, Dong et al. apply the Shapley value to attribute energy costs to apps on shared devices [30]. In networks, Feigenbaum et al. use cooperative games to attribute shared bandwidth costs during multicast transmission [34]. Han et al. use a repeated game that optimizes packet forwarding for strategic and distributed users [42]. Finally, in wireless networks, Saad et al. formalize time division multiple access (TDMA) as a cooperative game and develop distributed algorithms that direct users to better coalitions [93]. In contrast, we bring cooperative games to datacenter colocation and seek solutions that balance fairness, stability, and performance for strategic users.

2.7 Conclusions and Future Work

`Cooper` is a colocation framework that fairly attributes performance penalties, satisfies user preferences, and finds stable matches that are robust to strategic behavior. The framework employs sparse colocation profiles to predict preferences. It then employs preferences to find stable colocations in which no pair of strategic users would perform better by breaking away from the shared system. In addition to its game-theoretic properties, `Cooper` performs comparably with greedy and contention-minimizing mechanisms.

Extending `Cooper` to more than two co-runners and assessing stability guarantees is one of the future directions. In theory, stable matching for arbitrary group size cannot be solved in Polynomial time. Approximation algorithms exist for three co-runners under certain constraints[12]. In practice, a hierarchical approach could match applications and then match pairs. A clustering approach could classify applications into types and then match types. Stability guarantees in these heuristics may vary.

Stable matching is used in real, large-scale problems (e.g., assigning residents to hospitals, or students to schools). Microsoft has deployed stable matching in production systems for locality-aware scheduling [19]. In a similar spirit, `Cooper` operationalizes stable matching for future datacenters shared by strategic users.

2.8 Acknowledgments

We thank the anonymous reviewers of the 23rd IEEE International Symposium on High Performance Computer Architecture for their thoughtful comments and suggestions. We also extend special thanks to Janardhan Kulkarni, Brandon Fain, and Kamesh Munagala for their valuable discussions and insights.

This work is supported by the National Science Foundation under grants CCF-1149252 (CAREER), CCF-1337215 (XPS-CLCCA), SHF-1527610, and AF-1408784. This work is also supported by STARnet, a Semiconductor Research Corporation program, sponsored by MARCO and DARPA. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of these sponsors

3

Core Allocation with Market Mechanism

A datacenter allocates computational resources to applications based on static reservation or dynamic resource requirements. Processor allocations are inefficient when they are based on static reservations. For example, in public datacenters, users pay to reserve processors. These reservations are often conservative because users rarely know their needs at any given time, especially when applications have complex phases. In private systems, users pool resources together to construct a datacenter; these users are entitled to some fair portion of the combined resource. Such fairness incentivizes participation of strategic users, but harms system efficiency when users need less than their fair share yet retain exclusive access to those resources. This chapter addresses these inefficiencies in current fair resource allocation policies by providing accurate performance models to capture application's phase behavior and by designing core allocation mechanisms that can improve datacenter efficiency under the constraint of guaranteeing game-theoretic fairness.

We design a market for processors so that users can trade unused cores from their fair share on one server for additional cores on another. Users bid for cores based on parallelism in their applications. The market sets prices based on supply and demand for servers' cores. Responding to prices, users update bids. In market equilibrium, the market balances fairness and performance. Allocations are fair and guarantee sharing incentives, envy-freeness and Pareto efficiency. These game-theoretic desiderata are particularly important when strategic users compete for shared resources [23, 36, 119]. Furthermore, we show that the efficiency of these allocations is competitive with those from performance-centric policies.

The market's centerpiece is Amdahl utility, which we define to model the value of servers' cores. We develop a lightweight predictor that uses the inverse of Amdahl's Law to quickly estimates each application's parallel fraction. These fractions determine bids for servers' cores in each application phase and shape equilibrium allocations over time. Unsurprisingly, Amdahl utilities are well suited to modeling parallel performance for conventional, multi-threaded applications as well as emerging, task-parallel jobs. More surprisingly, our designed market mechanism with Amdahl's utility only requires closed-form, analytic equations for price and bid updating processes and can provide quick convergence for finding the market equilibrium during empirical experiments. Resulting allocations show satisfactory system performance and guarantee allocation fairness.

In this chapter, we make the following contributions:

- **Karp-Flatt Characterization.** We deploy and characterize Spark and PARSEC workloads. Amdahl's Law accurately models parallel performance and the Karp-Flatt metric, its inverse, accurately estimates an application's parallelizable fraction. (§3.2).

- **Prediction Framework.** We present a performance prediction framework for parallel applications with big datasets. We profile applications with small, sampled inputs and a few core allocations. We fit Karp-Flatt and Amdahl's Law for the effect of core count and linear models to estimate the effect of dataset size. Together, these models predict execution time for any core count and dataset size with on average 15% error (§3.2).

- **Market Mechanism.** We present a Fisher Market with Amdahl utilities. Users bid for cores based on applications' parallelizable fractions. The market collects bids and sets new prices based on supply and demand. We present a distributed mechanism to find a market equilibrium that performs well and guarantees game-theoretic desiderata (§3.3).

- **Performance and Fairness.** The market's allocations perform comparably to those from performance-centric mechanisms. In addition, the market guarantees sharing incentives, envy-freeness, and Pareto efficiency. Previously proposed markets perform worse or do not guarantee fairness. (§3.4).

3.1 Management Architecture

3.1.1 System Setting

We study the problem of allocating cores to strategic users in a private datacenter cluster. We consider a cluster with m heterogeneous servers shared by n users (Figure 3.1). Each user holds multiple applications. Applications have constraints or strong preferences over the servers. Application-to-server mappings are fixed before resource allocation. This system setting is different from the setting presented in Chapter 2. In Chapter 2, applications arrive the system with fixed amount of resource and the system manager colocates applications to mitigate colocation contention. Figure 3.1 presents a scenario where scheduling decisions happen before resource allocation and system manager dynamically adjusts allocated resource to improve system efficiency.

Such scenario could happen in a few cases. First, applications have hardware preferences. For example, in a heterogeneous datacenter with high-performance servers and low-power servers, high priority applications are usually matched to

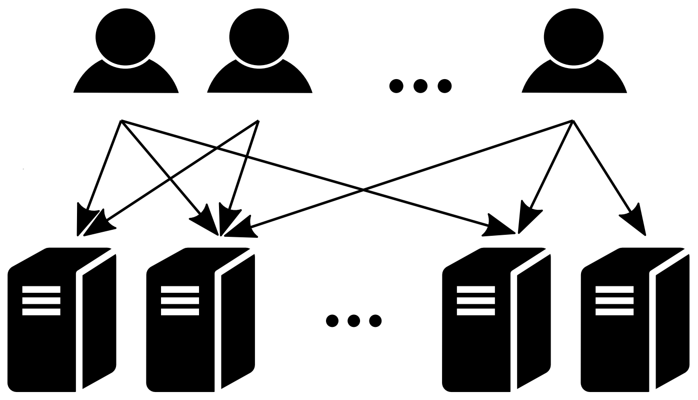


FIGURE 3.1: Cluster of m heterogeneous servers is shared by n users. Users are assigned to servers based on preferences or constraints.

high-performance servers to guarantee application performance during load spikes. Second, servers have configurations. Certain applications require special kernel versions or software libraries that are not installed on all servers in the datacenter. Third, data locality also plays the role of application-to-server mapping. Some applications are sensitive to data locality and always prefer to run on servers where the data is local. All above constraints can determine the application-to-server mapping.

Once the system manager assigns users' applications to specific servers, jobs assigned to the same server compete with each other for available cores on that server. We allocate cores to user's applications on each server dynamically. We answer the questions of "how many cores should be allocated to each application based on current load of the server and user's fair share of the datacenter".

3.1.2 Motivation

Fair policies may allocate processor cores inefficiently. In the case where each job is entitled to its fair share of a server's processor cores, the utilization of those cores may vary over time. Some computational phases have ample parallelism that can better utilize the cores whereas others do not. When the job needs less than its fair share, such allocations leave servers underutilized.

Work-conserving policies improve efficiency but compromise fairness. For example, dominant resource fairness (DRF) assigns cores and memory, but often leaves some fraction of a server's resources unallocated to guarantee game-theoretic properties [36]. Distributing idle resources to any user would sacrifice fairness for other users [23].

Suppose Alice and Bob each require computation for two jobs, which are assigned to two servers (i.e., four jobs on two servers). Alice's jobs require 2 and 8 cores whereas Bob's require 8 cores each. When each server has 10 cores, Alice and Bob are entitled to 5 cores on each server. Work-conserving allocations are efficient: Alice receives 2 cores on the first server and 5 cores on the second while Bob receives the remaining cores on each. Alice and Bob's total allocations are 7 and 13, respectively. Although efficient, these allocations do not incentivize Alice to share with Bob. In general, unfairness arises when a user donates some of its fair share of cores on one server, but is not guaranteed additional cores on another server. These users would have little incentive to share in the system with such management policies and would prefer receiving their static, equal shares of the cluster.

3.1.3 Allocation Architecture

We address the problem of allocating cores, fairly and efficiently, for multiple jobs on multiple servers. We design a management architecture based on two key ideas. First, Amdahl's Law describes a parallel job's utility from processor cores. Second, a market allows users to trade un-used cores from their fair allocation on one server for extra cores on other servers.

Within the market for processor cores, users are assigned fixed budgets that are proportional to their fair share of the system.[1] Each user is assigned servers for her

[1] Users could have different fair shares based, for example, on their hardware contributions to a shared system.

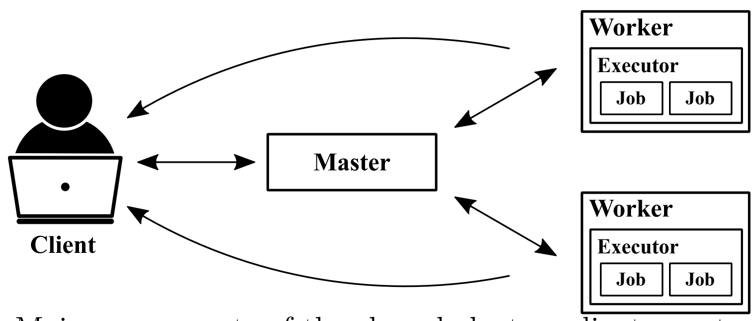


FIGURE 3.2: Main components of the shared cluster: client, master, and worker.

jobs. She bids for these servers' cores based on her jobs' demands and cores' prices. Bids reflect the importance of a server's cores for each user's workload on that server. Given bids, the system manager sets and announces new prices for cores. Then, users recalculate and submit new bids. The process repeats until no user changes her bid.

Figure 3.2 presents three main components in the market mechanism: clients, master, and workers. The master auctions processor cores on multiple servers. Master then apportions cores between jobs on servers by communicating with workers. Workers run users' jobs, monitor their execution, and supply logs to clients. The market works as follows: (1) Clients, on behalf of users, submit bids for processor cores on their assigned servers. If a client does not have a job on a server, her bid for that server is zero. (2) Master receives all bids before (3) setting and broadcasting new prices. (4) Clients recalculate bids based on new prices and submit updated bids. Prices and bids are updated until no client changes her bid. (5) Master sends processor core allocations to workers, and (6) each worker deploys cores for users' jobs.

Amdahl's Law. Clients analyze users' workloads and bid on their behalf. Specifically, each client uses Amdahl's Law to model utility (i.e., speedup) as a function of core allocation [8]. Amdahl's Law models execution time on one core, T_1, relative to the execution time on m cores, T_m. If fraction F of the computation is parallel, then speedup is:

$$s_m = \frac{T_1}{T_m} = \frac{T_1}{(1-F)T_1 + \frac{T_1 F}{m}} = \frac{m}{m(1-F)+F}. \tag{3.1}$$

Amdahl's Law assumes that the serial fraction cannot be accelerated with additional cores. It also assumes that the parallel fraction can be accelerated linearly with additional cores. Although these assumptions hold to varying degrees in real applications, Amdahl's Law is widely accepted and used for first-order approximations of speedup for parallel applications.

Each client in the market calculates her bid for cores using Amdahl's Law and parallel fraction F. Unfortunately, programmers rarely know exactly what fraction of their code or algorithm is parallel. For this reason, clients must empirically measure F with the inverse of Amdahl's Law, known as the Karp-Flatt metric.[2]

$$F = \left(1 - \frac{1}{s_m}\right)\left(1 - \frac{1}{m}\right)^{-1}.$$

According to Karp-Flatt Metric, we only need to measure speedup s_m to estimate F. But for which core count m should we measure speedup? If Amdahl's Law were perfectly accurate, then the answer would be "it does not matter." We answer this question, devising new methods for profiling performance and estimating an application's parallel fraction in §3.2.

3.2 Amdahl Utility

When using Amdahl's Law to model utility, we need to know an application's parallel fraction, F. Unfortunately, programmers may not know their code or algorithm's parallel fraction. In this section, we present an efficient method for profiling applications and estimating F using the Karp-Flatt metric, the inverse of Amdahl's Law.

[2] Karp-Flatt originally models serial fraction, $1 - F$.

Table 3.1: Cluster Specification

Item	**Specification**
Processor	Intel Xeon CPU E5-2697 v2
Sockets	2 sockets, NUMA node
Cores	12 cores per socket, 2 threads per core
Cache	32 KB L1 icache, 32 KB L1 dcache
	256 KB L2 cache, 32 MB L3 cache
DRAM	256 GB DRAM

First, we introduce the experimental methodology used for this part of the project (3.2.1). Second, we demonstrate the effectiveness Amdahl utility with a variety of parallel applications (3.2.2). Then, we detail a framework for predicting the metric (3.2.4). Finally, we evaluate prediction accuracy (3.2.5).

3.2.1 Experimental Methodology

Server. Table 3.1 describes the Intel Xeon E5-2697 v2 nodes used in our experiments. Each node has 24 cores (48 threads) on two chip-multiprocessors. The local disk holds application data. We deploy Docker containers [2] to run our applications with allocated core and memory amount, which ensures resource isolation. We use `cgroup` to allocate processor cores and memory to each container.

Workloads. Table 3.2 summarizes our PARSEC and Spark benchmarks [17, 118], which perform parallel computation on representative datasets. PARSEC benchmarks represent conventional, multi-threading whereas Spark applications represent emerging big data analytics workloads with data parallel engine. We run Spark applications in standalone mode (i.e., within single node). Each Spark application is comprised of multiple jobs. Each job is divided into stages and each stage has multiple tasks. The number of tasks in each stage usually depends on the size of the input data. For example, the first stage typically reads and processes the input dataset. Given Spark's default 32MB block size, a 25GB dataset (e.g., `webspam`) will

Table 3.2: Workloads and datasets

ID	Spark Name	Application	Dataset (Size)
1	`Correlation`	Statistics	webspam2011 [106] (24GB)
2	`Decision Tree`	Classifier	webspam2011 (24GB)
3	`Fpgrowth`	Mining	wdc'12 [5] (1.4GB)
4	`Gradient Descent`	Classifier	webspam2011 (6GB)
5	`Kmeans`	Clustering	uscensus [4] (327MB)
6	`Linear Regression`	Classifier	webspam2011 (24GB)
7	`Movie`	Recommender	movielens [3] (325MB)
8	`Naive Bayesian`	Classifier	webspam2011 (6GB)
9	`SVM`	Classifier	webspam2011 (24GB)
10	`Page Rank`	Graph Processing	wdc'12 [5] (5.3GB)
11	`Connected`	Graph Processing	wdc'12 (6GB)
12	`Triangle Counting`	Graph Processing	wdc'12 (5.3GB)
13	`Blackscholes`	Finance Analysis	native
14	`Bodytrack`	Computer Vision	native
15	`Canneal`	Engineering	native
16	`Dedup`	Enterprise Storage	native
17	`Ferret`	Similarity Search	native
18	`Raytrace`	Visualization	native
19	`Streamcluster`	Data Mining	native
20	`Swaptions`	Finance Analysis	native
21	`Vips`	Media Processing	native
22	`X264`	Media Processing	native

be partitioned into approximately 800 blocks. The Spark run-time engine creates one task to read and process each block. It then schedules tasks on available cores for parallel processing.

Profiling. For PARSEC applications, we use Linux `perf stat` to profile execution under varied cores allocations. For Apache Spark applications, we use Spark's event log file profile execution for jobs' stages under varied core allocations. We profile Spark with sampled datasets to accurately estimate an application's parallel fraction while reducing measurement overhead.

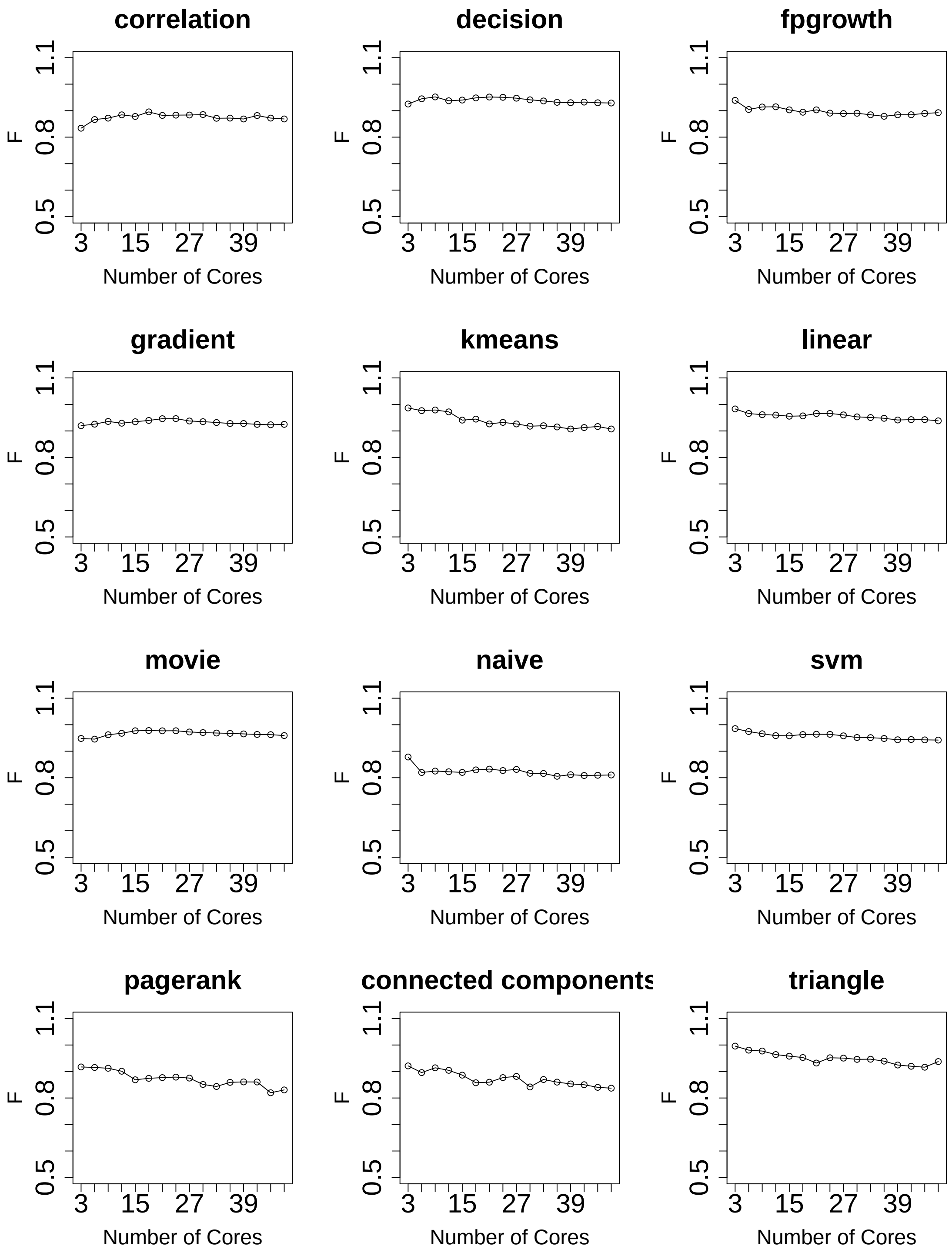


FIGURE 3.3: Calculated parallel fraction (F) for representative Spark applications as core count varies.

3.2.2 Demonstration of Karp-Flatt

Karp-Flatt Scaling for Core Count. According to Equation (3.2), we can calculate F from measured speedup s_c given c processor cores. However, Karp-Flatt does not specify for which c the speedup should be measured. If Amdahl's Law were perfectly accurate, the choice of c would not matter because the calculated F would be consistent for varied values of c's. But Amdahl's law is an approximation and calculated F differs for varied c's.

$$F_c = \left(1 - \frac{1}{s_c}\right)\left(1 - \frac{1}{c}\right)^{-1} \tag{3.2}$$

Figure 3.3 presents values of F that are calculated from varied measured speedups s_c and core counts c. For many applications, F is consistent across different core counts, which indicates that Amdahl's Law accurately models speedups. Amdahl's Law is less accurate for some applications because the parallel fraction F tends to shrink as core count increases. This trend indicates increased overheads such as communication, shared locks, and task/thread scheduling.

Figure 3.4 presents $\bar{F} = |c|^{-1} \sum_c F_c$, which is our expectation of an application's parallel fraction. This calculation averages F's over varied core counts. For our Apache Spark and PARSEC benchmarks, $\bar{F}$'s ranges from 0.55 to 0.99. Amdahl's Law is more useful when variance in F_c is smaller. Figure 3.5 presents $\text{Var}(F_c) = |c|^{-1}\sum_c (F_c - \bar{F})^2$, showing that Amdahl's Law and the Karp-Flatt metric accurately model parallel performance for most applications.

Limitations. Karp-Flatt is an accurate performance model for most applications, it falls short when overheads increase with core count. It is less accurate for graph processing (e.g., `pagerank`, `connected components`, `triangle`) since tasks processing different parts of the graph communicate more often as task parallelism increases. Karp-Flatt is also less accurate for Spark computation on small datasets that require

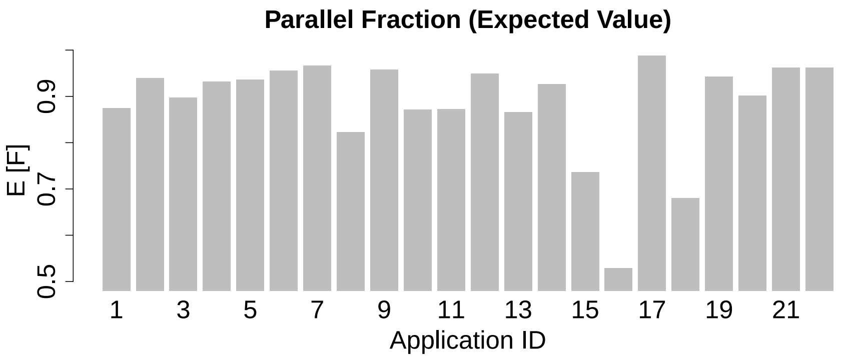


FIGURE 3.4: Expected parallel fraction $\bar{F} = \left(\bar{F} = |c|^{-1}\sum_c F_c\right)$.

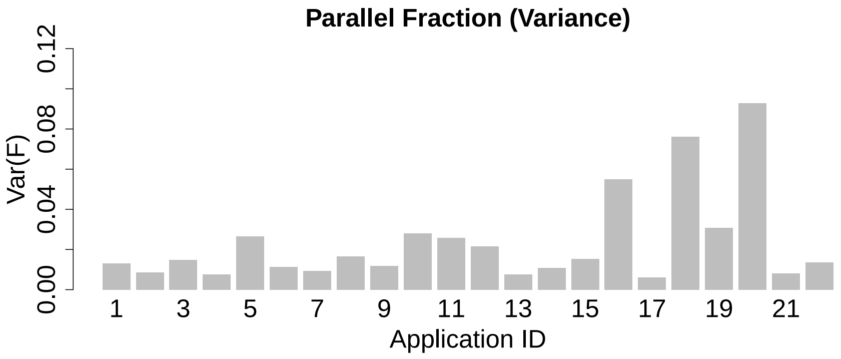
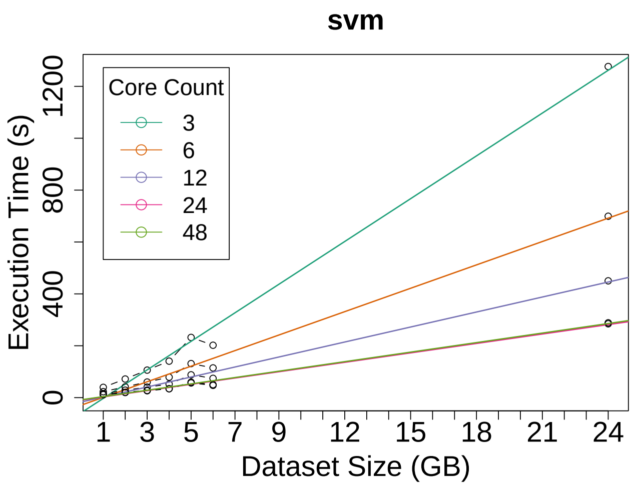


FIGURE 3.5: Variance in parallel fraction $\mathrm{Var}(F_c) = |c|^{-1}\sum_c (F_c - \bar{F})^2$. Lower variance indicates a better fit with Amdahl's Law.

few tasks (e.g., 11 tasks for `kmeans`) since adding cores rarely shortens execution time and often increases scheduling overhead. Karp-Flatt does not always work for traditional PARSEC applications (e.g., `dedup` for enterprise storage), because they are actively used for inter-thread communication and application's parallelism is largely affected by inter-thread communication.

3.2.3 Linear Scaling for Data Size

We estimate Amdahl utility and the Karp-Flatt metric with profilers. As computer architects, we typically reduce profiling costs by shortening the profile window (e.g., fast-forwarding to regions of interest) and reducing the dataset size (e.g., sim-small versus sim-large inputs). Similar strategies are required when profiling server appli-



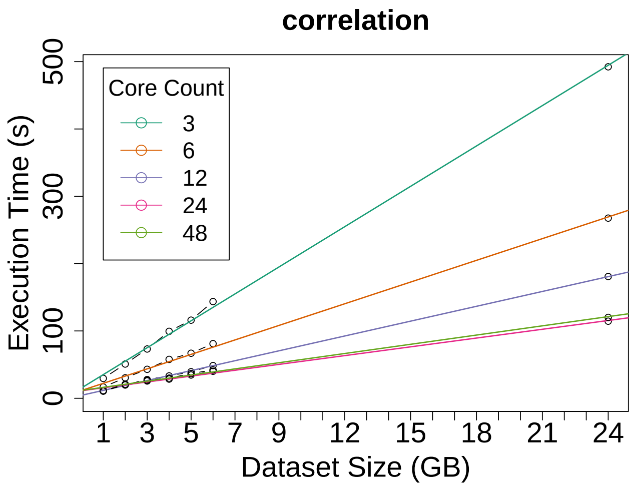


FIGURE 3.6: We sample the dataset size and measure performance for varied core allocations. We fit linear models to estimate execution time from dataset size. Data shown for representative applications `svm` (T) and `correlation` (B).

cations with large datasets. Unfortunately, we cannot efficiently profile performance on the whole dataset and a shortened profile window misses applications' long-term phase behavior.

Addressing these challenges, we quickly and accurately profile by reducing dataset size. We sample, uniformly and randomly, from the original dataset to create various smaller ones. Sampled datasets are small enough to complete computation, capture all program phases and explore all number of cores provided by servers. We measure application performance with sampled datasets and varied core counts, to obtain training data for the prediction framework.

Figure 3.6 indicates that execution time scales linearly with dataset size for presented applications. Therefore, a linear model can be fitted to sparse profiles and predict time required for any dataset size. We make the case for linear models with two representative applications, `svm` and `correlation`, computing on dataset sizes that include 1-6GB, 12GB, and 24GB.[3] Note that we require a different model for each core count. Models are more accurate when profiling with more cores (e.g., 48 cores).

Limitations. Note that Venkataraman et al. also use linear performance models to capture the effect of dataset size [103]. Although linear models work well for many applications, some require polynomial models because their execution time scales quadratically with dataset size (e.g., QR decomposition). We notice this effect during our model validation (§3.2.5); some applications show large prediction errors because of this assumption.

3.2.4 Prediction Framework

In this section, we present the proposed performance prediction framework illustrated in Figure 3.7. The framework consumes sampled datasets and core allocations to predict an application's parallel fraction and execution time. In the figure, prediction for parallel fraction and execution time progress horizontally and vertically, respectively.

- **Parallel Fraction F.** For a given dataset size, Karp-Flatt calculations predict the parallel fraction. The framework generates multiple predictions from varied, sampled dataset sizes and averages them.

[3] Note that sampled datasets could be smaller than these as long as the number of partitions, which dictate the number of tasks, is greater than the number of cores. Otherwise, there is insufficient parallelism.

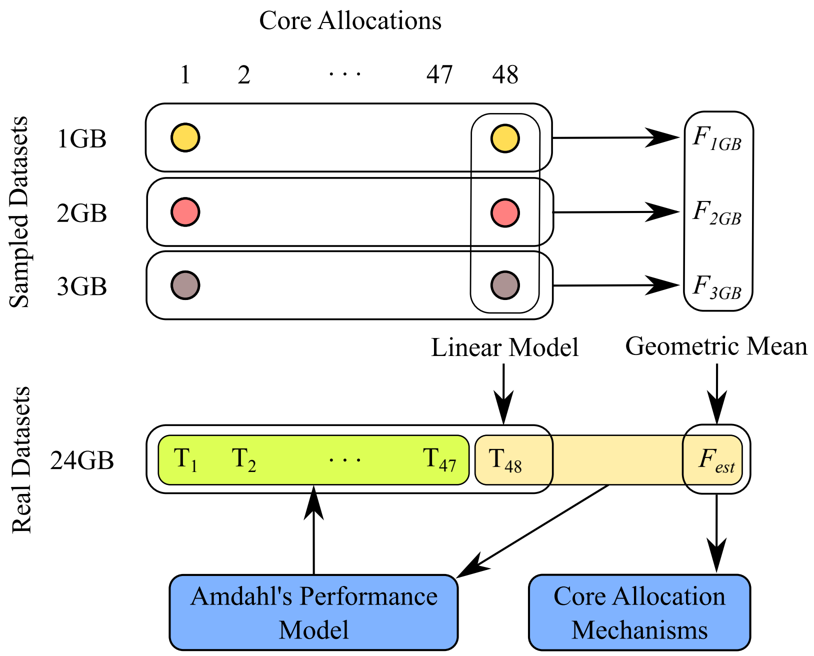


FIGURE 3.7: Prediction framework uses linear models to estimate the effect of dataset size and Karp-Flatt calculations to estimate the effect of core count. Estimates may be used to predict execution time (Amdahl's Law) or allocate cores (market mechanism).

- **Execution Time T.** For a given core allocation, linear models predict the application's execution time on any dataset size. The framework generates the prediction from a large core allocation (e.g., $c_{\max} = 48$) because profiling is faster and modeling is more accurate when given more parallelism.

Outputs from the prediction framework are broadly useful. Given $T_{c_{\max}}$ for some core allocation and F_{est}, we can use Amdahl's Law to estimate execution time for any core count and dataset size. Given F_{est} alone, we can implement various core allocation mechanisms including markets that use Amdahl utilities.

$$\frac{T_c}{T_{c_{\max}}} = \frac{c_{\max}}{F_{\text{est}} + (1 - F_{\text{est}})c_{\max}} \left(\frac{c}{F_{\text{est}} + (1 - F_{\text{est}})c} \right)^{-1} \tag{3.3}$$

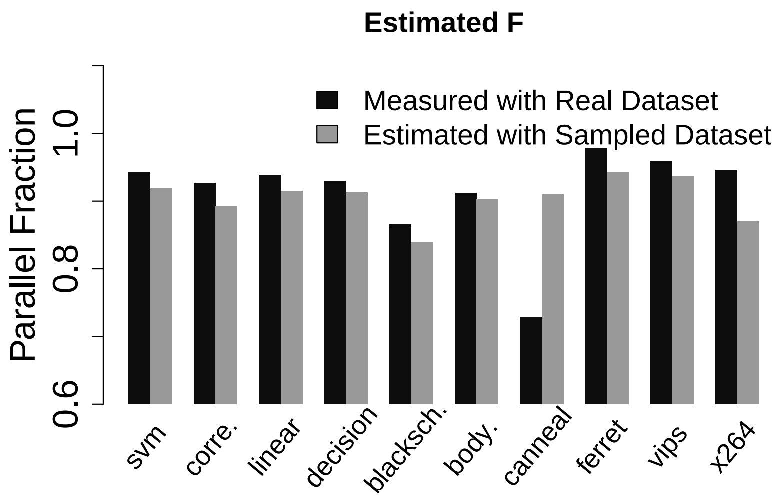


FIGURE 3.8: Accuracy of predicted parallel fractions F when using sampled datasets.

3.2.5 Prediction Validation

Figure 3.8 evaluates the accuracy of estimated parallel fractions F. Estimated values are calculated from small, sampled datasets. Measured values are calculated from the large, original dataset.[4] For most applications, errors are small and estimated values track measured ones. Relative accuracy is particularly important when using Amdahl utilities and allocating more cores to applications that benefit more. This means that if all estimated Fs are smaller than measured F by a similar factor, the F-based resouce allocation mechanisms will not be affected by the inaccuracy.

The `canneal` benchmark reports high prediction error because it is memory-intensive. Memory bandwidth utilization on small datasets may not be representative of that on larger datasets. If smaller datasets under-estimate memory bandwidth constraints, they also over-estimate speedups from additional cores. As a result, the estimated parallel fraction will be significantly larger than the one measured on the full dataset.

[4] For PARSEC, `simlarge` and `native` datasets produce estimated and measured values, respectively.

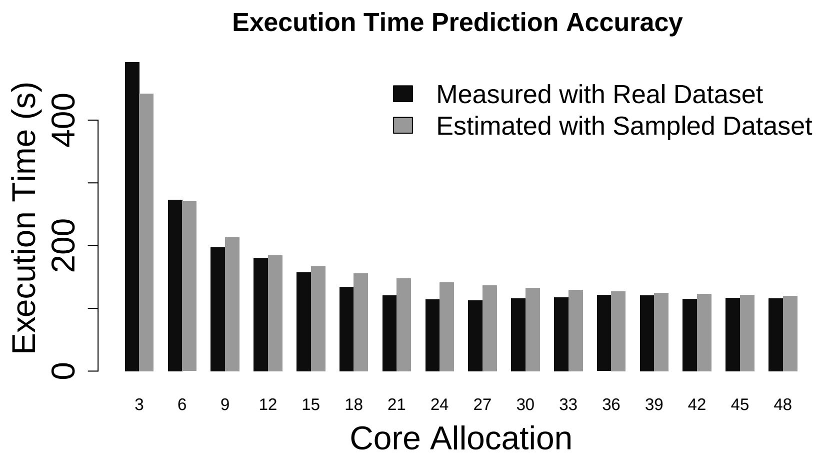


FIGURE 3.9: Accuracy of predicted execution time given varied core allocations. Data shown for `Decision Tree`.

Figure 3.9 evaluates accuracy when predicting execution time. Good predictions rely on accuracy for two estimates, one for execution time on the target dataset size and another for speedups on the target core allocation – see Equation 3.3. For the representative `Decision Tree` benchmark, we show accurate predictions for the target dataset and varied core allocations.

Figure 3.10 broadens the accuracy evaluation for varied applications. For each application, we present a boxplot to illustrate the range of errors when predicting execution time on varied core allocations. Our framework predicts execution with 5-15% error, on average, and 30% error in the worst case. As noted earlier, cache- or memory-intensive applications (e.g., `canneal`) are poorly modeled as small, sampled datasets cause the predictor to over-estimate the benefits of parallelism.

3.3 Market Allocation

Consider a datacenter with n users and m servers that hold varying numbers of processor cores; server j has C_j cores. A user runs multiple jobs and each job has been assigned to a server. The problem is allocating cores to users' jobs that have

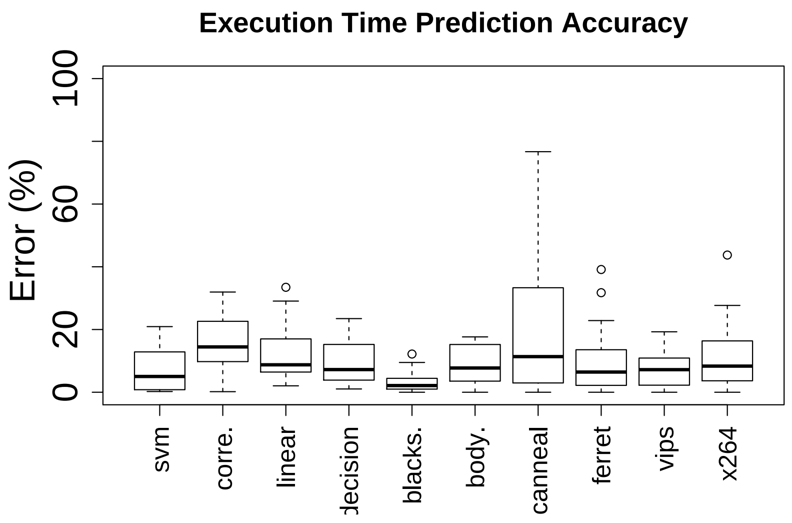


FIGURE 3.10: Accuracy of predicted execution time for varied applications. Boxplots show distribution of errors given varied core allocations.

been distributed across multiple servers so that the system performance is maximized under the constraint of game-theoretic fairness – sharing incentives, envy-freeness, Pareto efficiency.

3.3.1 Amdahl Utility Function

According to Amdahl's Law, allocating $x_{ij} \leqslant C_j$ cores to user i on server j will produce speedup s_{ij}.

$$s_{ij}(x_{ij}) \;=\; \frac{x_{ij}}{f_{ij} + (1 - f_{ij})x_{ij}}.$$

We define the Amdahl utility function as user i's utility from core allocations across m servers.

$$u_i(x_i) \;=\; \sum_{j=1}^{m} w_{ij} s_{ij}(x_{ij}).$$

Although Amdahl utility resembles a sum of speedups, it is actually a sum of work completed across multiple servers. Cores are allocated periodically and complete some quantity of work in each period (e.g., instructions or tasks). A job completes w_{ij} with one core and $w_{ij}s_{ij}(x_{ij})$ units of work with x_{ij} cores. User utility is total work across m servers.

Analytically, parameter w weights speedups obtained across multiple jobs. Although the parameter measures the baseline quantity of completed work, it could be further scaled to reflect system priorities. Higher priority could go to users who contribute more funds for the shared system or go to jobs that complete work more slowly.

Thus, Amdahl utility is consistent with system architects' views of performance. Its parameters model important determinants of performance – exploitable parallelism (f) and work completed (w). Moreover, the utility can weight work from specific users or jobs to reflect system priorities.

3.3.2 Market Mechanism

Problem Formalization. For the core allocation problem, we design a Fisher market[5] with participants described by Amdahl utility functions. The market has n participants and $j = 1, \ldots, m$ resources, each with capacity C_j. Resources are servers and capacity is the number of cores within a server. User i runs different jobs on different servers. Amdahl utility $u_i(x_i)$ describes jobs' benefits from cores and determines the user's preferences for allocations. Note that $x_i = (x_{i1}, \ldots, x_{im})$ specifies user i's allocation on each of m servers. User i bids for cores using her budget b_i. The market collects users' bids for each server and sets prices for cores. Given prices $p = (p_1, \ldots, p_m)$ for cores on m servers, users spend their budgets to buy their optimal allocations. For user i, this allocation will be a solution to the following optimization problem.

[5] In this section, we briefly describe the key mechanisms for the core allocation problem and omit detailed proofs. This part of the theory is developed by a collaborator and will appear with theoretical details in Seyed Majid Zahedi's Ph.D. thesis.

$$\max \quad u_i(x_i), \tag{3.4}$$
$$\text{s.t.} \quad \sum_{j=1}^{m} x_{ij} p_j \leqslant b_i.$$

Market Equilibrium. Users update their bids based on announced core price and market collects the new bids and reset the price of cores on each server. At equilibrium prices, the market clears and total demand meets total supply. In other words, all users receive their optimal allocations and there is no surplus or deficit of cores. Formally, price $p = (p_j)$ and allocation $x = (x_{ij})$ comprise a market equilibrium if the following conditions hold:

1. **Market Clears.** For each resource j, $\sum_{i=1}^{n} x_{ij} = C_j$
2. **Allocations are Optimal.** For each user i, allocation x_i maximizes u_i subject to budget $\sum_{j=1}^{m} x_{ij} p_j \leqslant b_i$.

We choose to use a market as the allocation mechanism because *the market equilibrium is efficient and fair.* As presented in above equations, in market equilibrium, all cores on every server is utilized, which maximizes system utilization. Moreover, from the perspective of individual user performance, allocations in market equilibrium are optimal for the presented utility function. Finally, market equilibrium provides sharing incentives, envy-freeness and Pareto efficiency. [22, 36, 119].

Finding the Market Equilibrium. A market equilibrium always exists because Amdahl utility is continuous and concave [13]. Proportional response dynamics (PRD) is a method that finds an equilibrium [111, 121]. Users examine prices and bid in proportion to utilities from resources. Next, the market sets new prices in proportion to bids for each resource. In response to new prices, users update bids.

PRD ends when bids and prices converge to stationary values. PRD is decentralized and does not require optimization as user and market responses require simple, proportional updates.

Market updates the core prices and users bids for cores based on the following updating rules. In iteration t, the price of cores on server j is the sum of users' bids divided by server capacity.

$$p_j(t) = \sum_i b_{ij}(t)/C_j.$$

Given these prices, user i's allocation of cores on server j is determined by $x_{ij}(t) = b_{ij}(t)/p_j(t)$. And her updated bid for the next round is

$$\begin{aligned} b_{ij}(t+1) &= b_i U_{ij}(t)/U_i(t) \\ U_{ij}(t) &= \sqrt{f_{ij} p_j(t)}\ w_{ij}\ s_{ij}(x_{ij}(t)) \\ U_i(t) &= \sum_j U_{ij}(t) \end{aligned}$$

We can prove, using KKT conditions, that any market equilibrium is a fixed point of this procedure and any fixed point of this procedure is a market equilibrium[6]. Finding an equilibrium for participants with Amdahl utilities is computationally efficient. Updates for market prices and user bids are expressed in closed-form equations. Updates are applied iteratively until the stationary. During empirical experiments, the procedure converges quickly (less than 200 iterations for 1000 users and 1000 servers).

[6] We omit detailed proofs because this part of the theory is developed by a collaborator and will appear with theoretical details in Seyed Majid Zahedi's Ph.D. thesis.

3.4 Evaluation

We evaluate proposed **Fisher Market (FM)**, which finds a market equilibrium as described in §3.3. Users use Karp-Flatt metrics to define Amdahl utilities. Users iteratively bid for processor cores on different servers based on utilities and prices, trading cores assigned to jobs with lower F's in exchange for cores assigned to jobs with higher F's. The bidding process terminates when the market clears (all cores are allocated) and allocations are optimal. Compared to performance-centric core allocation mechanisms, our primary goal is to guarantee game-theoretic fairness while preserving the system performance.

3.4.1 Baseline Policies

We compare FM against alternative allocation mechanisms, which are categorized into three classes — performance-centric, fairness-centric, market mechanisms. First, performance-centric mechanisms greedily allocate cores to the application that benefits most from an extra core, i.e., the application that has the largest marginal utility gain for the current core. **Greedy-I (G-I)** uses collaborative filtering to predict applications' execution time for different core allocations.[7] **Greedy-II (G-II)** replaces collaborative filtering with oracle performance. These mechanisms neglect fairness and provide upper bounds on performance.

Second, fairness-centric mechanisms define and enforce fair shares. **Equal Share (ES)** apportions a servers' cores to applications equally. ES guarantees game-theoretic fairness but neglects performance as it rigidly allocates a fixed share of cores to all applications without considering their utilities. **Proportional Fairness**

[7] We implement collaborative filtering in `R` using `recommenderlab` library. We profile performance for sampled core counts and treat utility as ratings. We use 20% sparse sampled rate as the input to. collaborative filtering. The recommender infers the complete performance matrix for all core counts and applications. Sometimes samples are insufficient to capture diminishing returns from Amdahl's Law. More samples are impractical but would improve accuracy.

(PF) apportions servers' cores in proportion to applications' estimated Karp-Flatt metrics, F_{est}.

Although PF may seem fair from single application perspective, it fails to guarantee users to get their endowed resource. If a users' applications all report relatively low F's, then she will not receive her fair share of cores.

Third, we compare against **XChange (XC)**, another market mechanism that balances fairness and performance. In XChange, a similar architecture is used for bidding and pricing; users iteratively bid for resources, the market announces new prices, and users optimize bids with hill climbing. Although XC finds a Nash equilibrium in theory, we find that hill climbing produces sub-optimal bids and the market fails to converge for Amdahl utilities in practice. Without the Nash equilibrium, XC loses its fairness guarantees.

We also evaluate **XChange-Modified (XC-M)**, which replaces hill climbing with the interior point method when optimizing bids [20]. Since Amdahl utilities are concave, the interior point method finds globally optimal bids in polynomial time. XC-M converges to Nash equilibrium, which puts a lower bound on efficiency and game-theoretic fairness.

FM differs from XC-M in several regards. First, FM's bidding process evaluates closed-form equations to update bids for new prices whereas XC-M's requires optimization (i.e., hill climbing or interior point). Second, FM produces a Market Equilibrium where users optimize their utilities under constraints, which guanrantees fairness from the perspective of envy-freeness, sharing incentives and Pareto efficiency. XC-M produces Nash Equilibrium, and in this case, it does not guarantee envy-freeness. Thus, FM provides stronger system properties, faster solution and lower implementation costs.

3.4.2 Experimental Methodologies

System Simulation. We use R-based simulations to evaluate performance, envy, and sharing incentives within large systems. For each allocation mechanism, we consider a population of users each of whom has a set of applications, sampled randomly from the pool in Table 3.2. When an allocation mechanism assigns cores to applications, we measure each application's performance on a real server to determine utility, envy-freeness, and sharing incentives.

Distributed Implementation. Although the experimental results presented in this chapter are collected from trace-based system simulation (§3.4), we do have an implementation of the market mechanism in real system to prove the feasibility. We implement the market mechanism, detailed in §3.3, with Scala. Our implementation uses the AKKA toolkit [1], which supports Actor-based concurrency. It includes clients, master, and workers with actors, each of which requires about 120 lines of Scala code. For a system with 1500 users, the distributed implementation finds the equilibrium core allocation in less than 5 seconds when clients are deployed with the master. When clients are deployed on a server other than the master, the network on the master becomes a bottleneck and convergence time increases to 60 seconds.

3.4.3 Performance

Figure 3.11 presents total system performance, which is defined as total utility of all users in system. Performance is normalized to that of G-I.

FM performs comparably to G-II and outperforms all other allocation mechanisms for every user population. G-II sets up an upper bound for the total system performance because it is a performance-centric mechanism and ignores fairness. G-II greedily assigns each core to the application with higher marginal utility given oracular knowledge of utilities. G-I predictions users' utilities with collaborative filtering, which has little intuition on the shape and structure of performance trends.

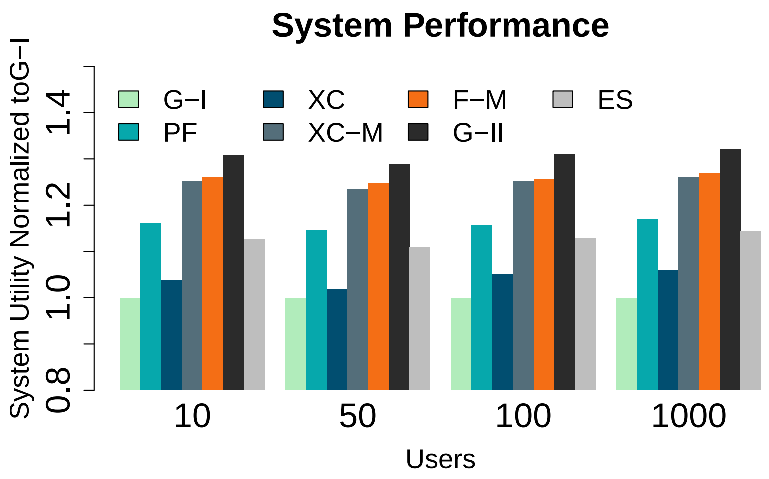


FIGURE 3.11: Performance measured in terms of total user utility. System grows with user population; n users request allocations on $m = n$ servers.

Predictions are inaccurate compared to those that use Karp-Flatt and, consequently, allocation performance suffers.

ES has low system performance as it fails to adapt core allocation to applications' diverse parallelism and assigns a fixed share to all. PF considers these differences and allocates cores in proportion to applications' parallelizable fractions. Yet, PF performance suffers because it must allocate proportionally to F even though performance increases non-linearly with respect to F, according to Amdahl's Law. PF tends to over-allocate for applications with low F and under-allocate for those with high F, leaving performance unexploited.

XC users optimize their bids with a hill climbing heuristic [108, 109]. But for Amdahl utilities, the heuristic almost always terminates without finding the optimum. Suboptimal bids degrade system performance. XC-M solves this problem, using the integer point method to optimize each user's bid in response to other users' bids. However, the resulting Nash Equilibrium is not necessarily a Market Equilibrium when users are price-anticipating.[8] Price-anticipating users can manipulate their

[8] With price-taking users, the Nash Equilibrium is the same as the Market Equilibrium.

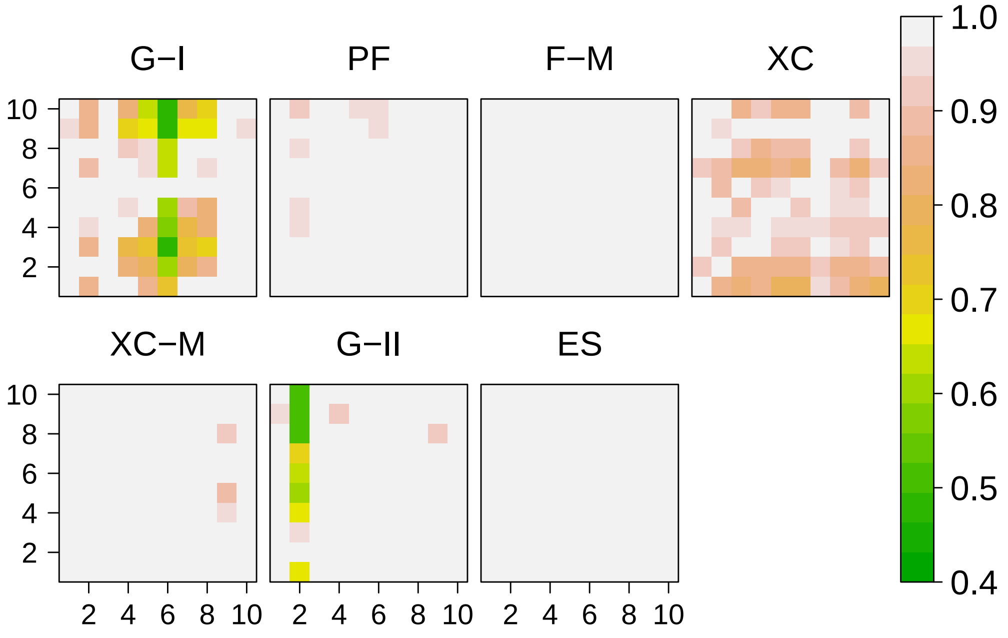


FIGURE 3.12: Envy-free Index - Darker colors mean greater envy. X-axis and Y-axis show 10 users. Matrix items indicates how much these users envy each other (pairwise envy). Simulation parameters: $n_{user} = 100$, $n_{appPerUser} = 10$, $n_{server} = 100$.

bids to affect the price and degrade system performance.

Note that XC and XC-M both require users to run a polynomial time optimization algorithm to find their bids in each allocation round. In contrast, FM uses a closed-form formula to update users bids in each round, which has significantly lower implementation and computation overheads and can be scale well to large systems.

3.4.4 Envy-Freeness

A mechanism is envy-free when no user prefers any other user's allocation more than her own. Envy-freeness is a key outcome for fair resource allocation. We use a pairwise envy index to quantify user i's envy towards user j. Equation (3.5) describes envy-free index. For every pair of users i and j, the envy-free index compares user i's utility for her own allocation, $u_i(x_i)$, to its utility for user j's allocation, $u_i(x_j)$. If $u_i(x_i) < u_i(x_j)$, then the index is less than one and user i envies user j. The index is always less than one and a smaller index means greater envy.

$$\text{Envy-free Index} = e_{ij} = \frac{u_i(x_i)}{\max\{u_i(x_i), u_i(x_j)\}} \tag{3.5}$$

Users typically do not envy each other when the number of users' applications is small relative to the number of servers in the system. For example, consider a system with 100 servers and two users, each with 5 applications to run. The probability that any server is assigned to both users is very low. Without colocated computation, the users have no utility for each others allocations.

However, when the number of users' applications approaches the number of servers, users could envy each other and the choice of allocation mechanism matters. To study envy, we group users and assign the same set of servers to each user in a group. Users within a group could envy each other as each has non-zero utility for cores on all servers.

We evaluate a system with 100 servers, 100 users and 10 applications per user. We group users into 10 groups and assign the same set of servers to the users within the same group. Figure 3.12 presents the pairwise envy index for the group with greatest envy (ie., lowest index). Square (i, j) indicates user i's envy towards user j's allocation. ES is envy-free because equally dividing resources ensures that no user prefers another's allocation. FM is the only other mechanism that theoretically guarantees envy-freeness. In practice, however, FM occasionally reports one or two users in a group with slight envy. We deviate from theory for two reasons. First, FM guarantees envy-freeness assuming Amdahl utilities perfectly predict system performance – envy may arise when utility deviates from physical system measurements. Second, envy may arise from the rounding process, which converts fractional allocations into integer allocations.

G-I and G-II report the greatest envy, which is not surprising because they pursue performance and neglect fairness. XC also reports significant envy as the hill climbing algorithm fails to approximate the market's Nash equilibrium. XC-M reports less envy than XC and achieves the Nash equilibrium's theoretical guarantee on envy (0.828-approximate envy-free [109]). Finally, PF defines a different notion of fairness that neglects envy.

3.4.5 Sharing Incentives

Allocations incentivize sharing when users prefer their allocation over static, equal shares. Without sharing incentives, system may fragment and strategic users have no incentive to participate in large shared systems, which hards system integrity and efficiency. Equation (3.6) introduces the sharing index as the metric for quatifying sharing incentives. Sharing index is the ratio between a user's utility under the current mechanism to her utility under equal shares (ES). A larger index means greater sharing incentives. Then, the sharing index of a resource allocation mechanism is the minimum of these ratios across all users. A mechanism satisfies sharing incentives if the index is greater than one.

$$\text{Sharing Index} = \min_i \left[\frac{u_i(\text{mechanism})}{u_i(\text{static})} \right] \tag{3.6}$$

Figure 3.13 presents sharing index for varied user populations and allocation mechanisms. FM always reports a sharing index greater than one. Moreover, its sharing index is greater than that of all other mechanisms under various system sizes, indicating that FM does most to incentivize users.

Performance-centric mechanisms, G-I and G-II, do not satisfy sharing incentives. G-II achieves high overall system performance, but does so at the expense of low-utility users. These users are not incentivized to share with others. Strategic users would opt out from such unfair management policies and may build a smaller, private system to improve their performance.

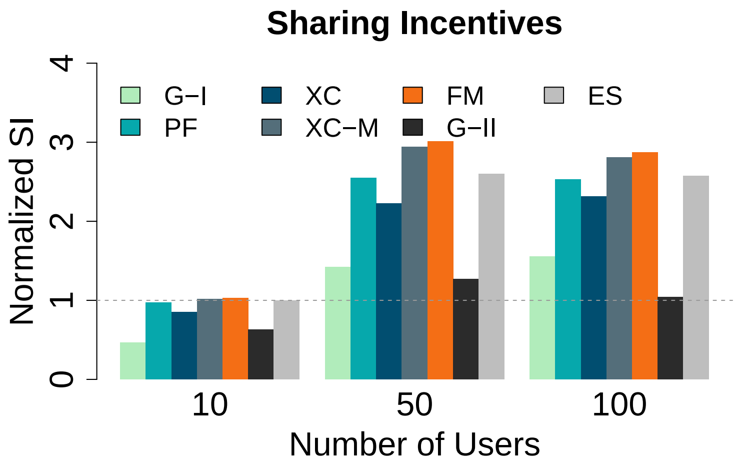


FIGURE 3.13: Sharing Incentives

XC, XC-M, PF and ES all incentivize users to share in big systems. Sharing index increases as the number of servers increases. In large systems, especially when users do not have applications on most of the machines, their utility for these servers becomes zero under static, equal shares. Therefore, it is much easier to satisfy sharing index greater than 1 when normalized to ES because ES perform badly for such sparse case. Nevertheless, for the case where users have applications on every server (i.e., number of users equals 10 case), FM still has the highest sharing index.

3.5 Related Work

Algorithmic Economics for Datacenter Management. Recent studies in the architecture and systems communities have explored economic game theory for managing datacenter resources [32, 36, 63, 82, 85, 119]. Ghodsi et. al propose Dominant Resource Fairness (DRF) to allocate cores and memory for applications sharing a chip-multiprocessor [36]. Zahedi et al., proposes Resource Elasticity Fairness (REF) using the Cobb-Douglas utility function for allocating cache capacity and memory bandwidth [119].

DRF and REF argue that the resource allocation policies in datacenters should guarantee game-theoretic desiderata (Sharing Incentives, Pareto Efficiency, Envy-Freeness, and Strategy-Proofness) to guard against strategic users and encourage user participation. Our market allocations processor cores to produce a market equilibrium and guarantee the same desiderata.

Markets for Resource Allocation. Economic theory, in particular, casts resource allocation as a market allocation problem and asks users to bid for resources [16, 39, 108]. Guevara. et al proposes an auction-based mechanism to allocate heterogeneous cores to web search queries[39]. Wang et al. proposes XChange [108], a trading market with a price setting mechanism for core and memory in a chip multiprocessor. XChange, the most related piece of work, provides a Nash Equilibrium that does not guarantee Pareto Efficiency and Envy-freeness. Our mechanism produces a Market Equilibrium and guarantees game-theoretic desiderata.

Datacenter Resource Management. Cloud infrastructure (e.g. Amazon EC2) uses a reservation system and expects users to provide the required resources (e.g., number of cores, amount of memory, number of VMs). Hindman et al. propose Mesos [44], which implements a request-grant abstraction among different parallel frameworks (MPI, Hadoop, Spark). These reservation-based systems rely on users to report resource usage, burdening users and introducing opportunities for strategic action.

We propose fast and accurate performance models to calculate the Karp-Flatt metric and fit the Amdahl utility function. Moreover, we design a mechanism that produces a market equilibrium given Amdahl utilities. Compared with previous frameworks that include sophisticated performance models (e.g., Paragon[27], Quasar[28] and Bubble-up[70]), our performance model is simpler yet accurate. And we propose a sophisticated allocation market that guarantees fairness yet performs well.

3.6 Conclusions and Future Work

We propose a Fisher market to allocate cores among shared users in a private datacenter. In the market, we model users' utilities of their parallel applications with Amdahl's Law and we deploy a lightweight performance prediction framework to estimate the parallel fraction (Karp-Flatt Metric) of applications with big datasets. We validate our performance model with real system measurements and achieve on average 15% prediction errors for runtime predictions. More importantly, we develop an allocation framework with distributed bidding mechanism and centralized price updating mechanism to drive the market to an equilibrium. We use system simulations to quantify the performance and fairness tradeoffs for resulting allocations and compare with state-of-art resource allocation mechanisms. Our coordinated design of performance models and allocation mechanism provide comparable system performance to traditional performance-centric policies; we outperform state-of-art market mechanism and fairness-centric policies in terms of performance, sharing incentives and envy-freeness.

Future works can extend the market mechanism to utilities that are different from Amdahl's Law. During the performance model validation, we discover that Amdahl's Law does not work well for certain types of applications, e.g., graph processing, I/O intensive and memory intensive applications. This is mostly because of two reasons. First, application parallelism changes when increasing dataset size. A linear scaling model will not capture such trends, instead quadratic and exponential models should be used. Second, such applications' performances do not scale linearly on core count, which means Amdahl's Law is not an accurate performance model. For memory intensive and I/O intensive applications, cores wait for I/O or memory requests instead of doing useful work. Therefore, adding more cores does not improve the performance linearly. In these cases, we need to develop more complex performance models to better capture application's diverse phase behavior. However, such complexity poses great challenges in market mechanism design, i.e., convergence to the market equilibrium is hard to guarantee. We believe that certain utility functions will be able to accurately model applications' performance and quickly reach market equilibrium convergence.

3.7 Special Acknowledgement

In this project, the author was primarily responsible for Amdahl's Law performance models and system simulations. This included several important notions, including demonstration of the chosen performance model with various emerging applications, designing and validating performance prediction framework for application parallel fraction and application runtime. The author was also responsible for conducting experiments to evaluate the performance, envy-freeness and sharing-incentives of the market mechanism. Seyed Majid Zahedi developed the theory for the market mechanism with Amdahl's utility. Amir Rahimzadeh Ilkhechi implemented a distributed framework in a small scale (1 node) to demonstrate the feasibility of the market in real system. In summary, collaborations were interactive and all authors had contributed to every part of the project.

4

Modeling Communication Costs in Blade Servers

Datacenters demand big memory servers for big data applications. Blade servers provision abundant memory in a dense form factor and their distributed shared memory are well suited to big data applications. Researchers have prototyped or emulated blade architectures [46, 61] to understand their potential and to demonstrate key capabilities, such as fine-grained access and address translation. Beyond specific designs, however, researchers must assess sensitivity to technology parameters and explore server organizations. Unfortunately, existing experimental methods lack the required flexibility.

In this chapter, we present technology models that enumerate communication paths through a blade server and identify contributors to delay and energy. In addition to DRAM costs, we account for inter-processor and inter-blade data transfers. With these models, researchers can explore scenarios in system organization and data movement, identifying opportunities and addressing challenges.

One particular challenge is non-uniform memory access (NUMA). In a blade server, a processor can retrieve data from memory via several communication paths that introduce multiple levels of NUMA. Conventional wisdom argues that NUMA harms performance and should be avoided [98]; prior work has proposed task and data managers that restrict remote memory access [37, 59, 65, 88]. However, NUMA might improve performance; a system that selectively permits NUMA might dequeue tasks sooner and increase throughput. Technology models help researchers understand the benefits of permitting NUMA and develop new schedulers that balance latency and throughput.

Our interdisciplinary study integrates insights from interconnect technologies, queueing theory, and big data applications. Herein we provide new methodologies for coodinating the design of software management policies (e.g., schedulers) and system organizations (e.g., inter-connection technology and topology). The proposed method based on system simulation and parameterizable models can reduce the design costs of datacenter hardware organization and configurations and can provide short optimization cycles for performance and energy goals.

This chapter is organized as follows. In §4.1, we motivate the study of blade servers and present related works on scheduler design to mitigate non-uniform memory access. In §4.2, we present technology models that enumerate communication paths through a blade server. In §4.4, we introduce the methodology and platform to performance coordinated study on scheduler design and technology models. In §4.5, we demonstrate the ability of our models with the case study of managing non-uniform memory access.

4.1 Background and Motivation

4.1.1 Modeling Blade Servers

Blade servers provide compute and memory capacity in a dense form factor. Multiple processors reside on a blade, communicating via high-speed serial links and sharing a physical address space. Multiple blades reside on a backplane, which also permits communication. Together, advances in blade architectures and I/O interfaces increase capacity, enable sharing, and improve flexibility for big memory systems.

Capacity. Disaggregated memory architectures increase capacity by decoupling the provision of compute and memory resources [61]. Whereas traditional servers increase memory capacity by adding processors and their attached memory channels, disaggregated servers add memory blades that are designed to supply shared capacity. Architects can easily adjust memory-to-capacity ratios by tailoring the memory blade to workloads.

Sharing. Hardware advances in I/O bridges and switches enable sharing for memory. Memory controllers identify accesses to remote memory on other blades [81], PCIe bridges expose I/O interfaces on each blade, and PCIe switches route packets between blades within the server [46, 50, 89]. These communication paths connect disparate physical address spaces and permit a processor to access another blade's memory. Sharing is particularly useful for task parallel computing with heterogeneous tasks.

Whereas prior studies prototype or emulate a specific implementation to prove a concept, we present generalizable models for blade servers. Models enable design space exploration for hardware organizations, software execution models, and system management strategies. We model multi-level NUMA, looking beyond coarse "local" and "remote" labels to account for memory, inter-processor links, and inter-blade links. Moreover, we model communication energy for the first time for blade servers and compare the efficiency of remote access versus data migration for big data applications.

4.1.2 NUMA Systems Setting

Many concepts contribute to build large shared memory systems, e.g., partitioned global address space, software distributed shared memory, cache-coherent memory, user-level message passing and remote direct memory access. In this section, we make the case for two big memory system settings, cache-coherent NUMA system and scale-out NUMA system, and discuss the benefits of using them for emerging big data applications (i.e., Spark [118], and GraphLab[66]).

Cache-coherent NUMA system (ccNUMA). ccNUMA provides cache-coherent global physical memory space. AMD and Intel hardware provide cache-coherent physical memory sharing across nodes through inter-processor technology (HyperTransport[47] or Intel QPI[49]). Direct memory access can be performed over the global address space with secure cache-coherence protocols, and can be implemented by hardware. In [46], Hou et al. propose a blade system with coherent memory sharing through software, guaranteeing a single writer multiple reader invariant. Such blade system provides coherent address sharing across processors and blades, becoming a potential hardware for in-memory computational frameworks, e.g., Apache Spark.

Spark provides fast iterative MapReduce computation by caching intermediate results between computational stages in memory with resilient distributed datasets (RDD). Spark can benefit from big memory systems because tasks more often find data in memory. However, Spark workers can potentially find RDDs being cached into a remote memory region on a NUMA machine, causing delays in individual tasks. Since individual tasks are extremely small and only touch kilobytes-size pages, task and data migration happen rarely. The operating system migration policies (e.g., first-touch [104], next touch [65]) benefit little from such short-lived tasks and frequent task migration could potentially cause congestion traffic in memory systems. Therefore, the Spark task scheduler should be aware of NUMA effects to improve task memory locality.

Scale-out NUMA system (soNUMA) Novakovic et al. proposes Scale-out NUMA in [81], a distributed shared memory system with hardware support. It provides fast, fine-grained direct memory access (load and store) to remote shared

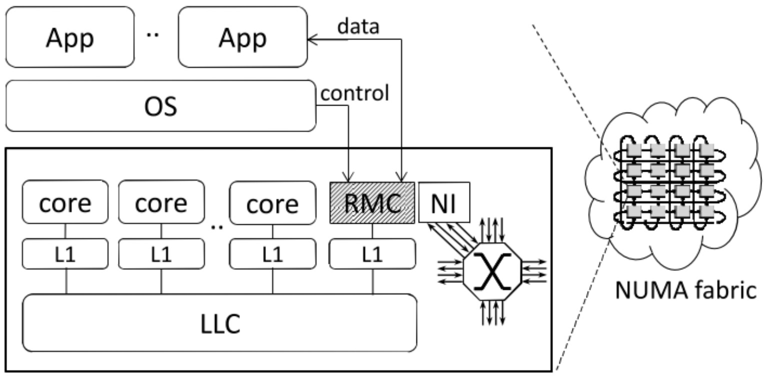


FIGURE 4.1: Scale-out NUMA architecture [81].

memory with a remote memory controller (RMC). Figure 4.1 illustrates one soNUMA node. In soNUMA, each node runs an instance of an operating system. The cores on the same node share the last-level cache and memory system. The RMC bypasses cores and the OS kernel; it directly uses network interface to access remote memory regions. soNUMA differs from software distributed shared memory (DSM) because it provides support for fine-grained remote direct memory access with hardware (RMC), which makes it order of magnitude faster than DSM. Software DSM relies on OS page-fault and relaxed memory model to propagate the results.

We believe future hardware for big data applications will move towards the soNUMA style, supporting fast, fine-grained direct memory access within large distributed memory system. In soNUMA, NUMA effects still exist and memory locality-aware schedulers will play a role to improve the system efficiency.

4.2 Modeling Communication Cost

In this section, we identify key architectural parameters for blade servers and derive cost estimates for three types of communication channels in a blade architecture: (1) memory bus, (2) inter-processor links, and (3) inter-blade links. By simulating a server with these estimates, we can assess a task's NUMA tolerance and commu-

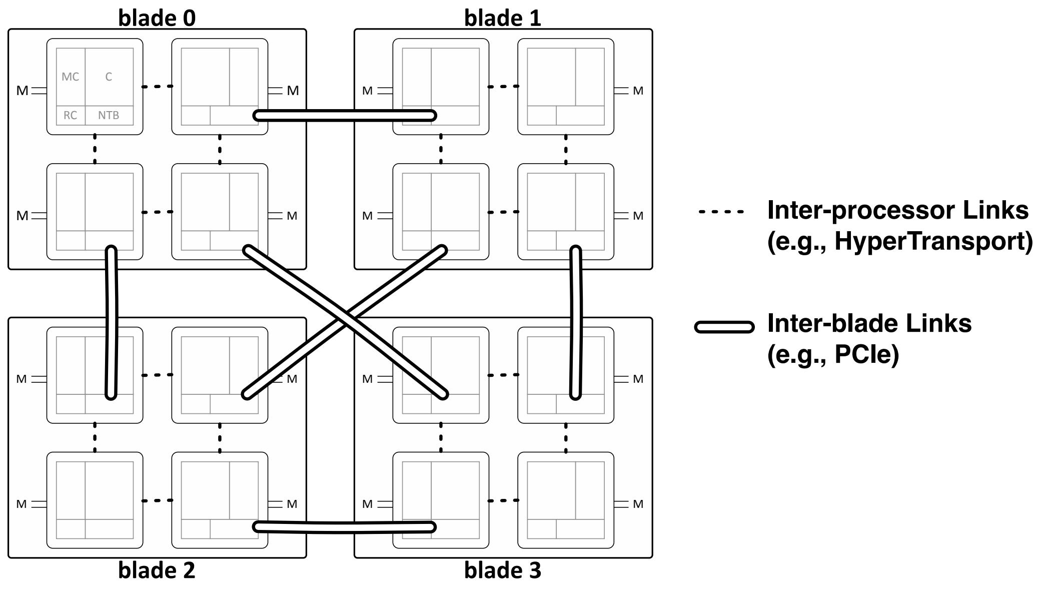


FIGURE 4.2: Blade architecture with sixteen-core processors, four-processor blades, and a four-blade server.

nication efficiency. We analyze communication scenarios for a representative system (Figure 4.2), which includes sixteen-core processors, four-processor blades, and four-blade servers. The system is representative of high-performance servers and is just one of many possible organizations. Our models can also be used to performance design space exploration for blade organizations (future work).

4.2.1 Memory Communication

The memory controller is integrated into the processor die and responds to last-level cache misses. The controller translates a memory request into activate, read, write, and precharge commands that traverse the memory bus to access the DRAMs. If the DRAM buffer holds the requested row, the controller issues reads and writes immediately. Otherwise, it first precharges the buffered row and activates the requested row [53].

Delay. Memory latency is affected by array latency, queueing delays, and request scheduling. The controller can schedule memory requests for a mix of latency,

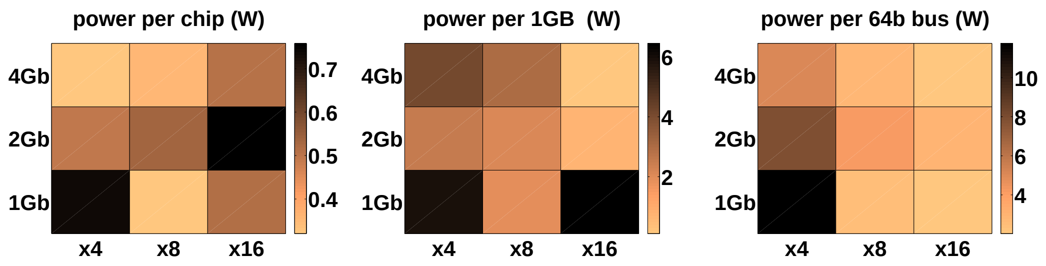


FIGURE 4.3: DRAM power (a) per chip, (b) for 1GB of capacity, (c) for 64b interface. Estimates from Micron [75].

throughput, and fairness targets [78, 79]. For example, the controller may re-order requests, allowing a later request to access buffered data even if doing so delays an earlier one. Sophisticated scheduling mitigates row buffer misses and bank conflicts, which add to memory latency. Simulators precisely capture scheduling and timing effects [91]. Typical DRAM latency is 50 to 100ns.

Energy. Energy is consumed by the DRAM core and the chip interfaces. The core consumes dynamic energy due to precharges, activates, reads, and writes [105]. DDRx interfaces, which include delay-locked loops and on-die termination for signal integrity, draw static current regardless of channel utilization. For this reason, high-performance memory systems are energy-disproportional for tasks with modest demands for memory bandwidth [68]; alternative chip interfaces can improve efficiency [41, 69].

DRAM is available in several pin and capacity configurations. If a memory system uses low-capacity DRAMs with narrow interfaces, it will use many chips and power increases. However, using high-capacity DRAMs with wide interfaces complicates mechanisms for reliability [68, 100, 114]. Figure 4.3 presents representative power numbers. 2Gb DDR3 with 8DQs strikes a balance and dissipates approximately 2W per GB. A typical server has 8GB per channel, two channels per processor, and four processors per blade; 64GB of DRAM dissipates 128W.

4.2.2 Inter-Processor Communication

Multiple processors are integrated into a blade. They share a physical address space and support coherent access to shared memory. Both the local memory controller and the interconnect controller observe a last-level cache miss. The latter uses the memory address to identify and route a memory request destined for another processor's controller. A remote memory request may require one or two hops, which introduces latency and energy costs in addition to those imposed by DRAM.

Delay. Interconnect and protocol dictate the costs of inter-processor memory access. The interconnect uses serial, point-to-point links for high data rate and low latency. The interconnect implements a protocol, appending headers and encoding the memory request to create a packet for transmission. Packet construction and transmission increases memory access time.

We use HyperTransport [45] for the inter-processor connection. HyperTransport transmitter logic requires 18ns to encode contents and add headers. Packet transmission and link interface circuitry adds 14ns. Receiver logic requires another 18ns. In total, the packetized request requires 50ns to reach a remote memory controller. With round-trip overheads and DRAM access delay, remote data access requires at least 150ns.

Energy. To assess energy costs, we examine serial links and their interfaces. A serial link requires serializer/deserializer (SerDes) circuitry at its endpoints to convert data between parallel and serial interfaces. This circuitry determines the energy cost per bit transferred. How this cost translates into system power depends on the number of processors, the number of links between them, and the data rates of those links [41].

HyperTransport links connect two processors. The connection is 16 lanes wide and each lane is implemented with serial links. Differential signaling requires paired links. And bi-directional communication requires two paired links since a serial link is unidirectional. Thus, 16 lanes requires 64 links and 128 SerDes. Each SerDes consumes 10pJ per bit transferred and can transfer up to 6.4Gb per second [56, 110]. We estimate power by multiplying the number of interfaces, the transfer rate, and the cost per transfer – power is 8.2W per path and servers may use five paths dissipating up to 40W.

4.2.3 Inter-Blade Communication

Multiple blades are integrated into a server. Blades share an address space in a distributed shared memory machine and communicate via a backplane interconnect. Bridges and switches perform address translation and route memory requests to the appropriate blade [46, 57, 89]. We describe a blade architecture in which a processor can access a remote blade's memory with load/store instructions via PCIe.

PCIe Interfaces. The memory controller, PCIe root complex, and non-transparent bridges (NTBs) are all integrated into the processor die [50]. The BIOS configures PCIe ports to serve as NTBs, which provide blade-to-blade interfaces. The BIOS also divides the physical address space into regions, one for each blade.

When a last-level cache miss occurs, the memory controller uses the memory address to differentiate local and remote memory requests. The PCIe root complex handles a remote request, creating a packet that travels to the destination blade via serial PCIe links. The receiving blade's NTB translates the request's address into its own physical address space, using BIOS-configured registers that track address space regions.

Delay. Communication costs are determined by the interconnect, which uses serial links. Each inter-blade connection begins and ends with a bridge that performs address translation. A typical 64b PCIe transmission requires 240ns of which 50ns is attributed to DRAM [45]; the remaining 190ns is round trip transmission delay for

PCIe links.

In a blade server system, the cost of inter-blade communication also depends on the number of HyperTransport hops. Six PCIe links are used to connected twelve NTBs for the example in Figure 4.2. A memory request may traverse HyperTransport to reach the correct bridge and traverse PCIe to reach the correct blade. We calculate the expected HyperTransport delays on sending and receiving blades, which are incurred in addition to PCIe and DRAM latencies. With round-trip overheads, accessing data on a remote blade may require 410ns, including 50ns for DRAM, 190ns for PCIe, and 170ns for expected HyperTransport delay.

Energy. Energy costs increase with blade connectivity. In our example, each uni-directional connection is 16 lanes-wide and a bi-directional connection requires 32 lanes that dissipate 4.8W. Each lane dissipates 150mW and transfers data at 0.5GB/s [58]; energy cost is 37.5pJ/bit. Moreover, bridge logic and SerDes circuitry dissipate 2.5W. In total, an inter-blade link dissipates 10W for 32 lanes and two bridges.

We consider connectivity between four blades, each with four processors and integrated bridges. These NTBs support back-to-back PCIe connections between pairs of blades [50]. A fully-connected network requires six connections and twelve NTBs, which in total dissipate up to 60W.

4.2.4 Summary and Discussions

Table 4.1 summarizes our estimates of communication costs and Figure 4.4 illustrates the distribution of power consumed by data communication. DRAM accounts for much of the communication power, but inter-processor and inter-blade overheads still comprise nearly 30% of the total. More generally, memory and data account for more than 25% of datacenter energy [14].

In summary, our analytically derived technology models are accurate and con-

Technology	DDR3 8 2Gb,x8	HT3.1 PTP	PCIe 2.0 PTP
Data Rate (GB/s per lane)	0.2	0.8	0.5
Lanes	x64	x32	x16
Uni-directional B/W (GB/s)	12.8	25.6	8.0
Transfer Energy (pJ/bit)	160.0 (@20% util.) 70.0 (@100% util.)	36.0	37.5
Latency (ns)	50-100	100 RT	190 RT

Table 4.1: Summary of technology models and estimates

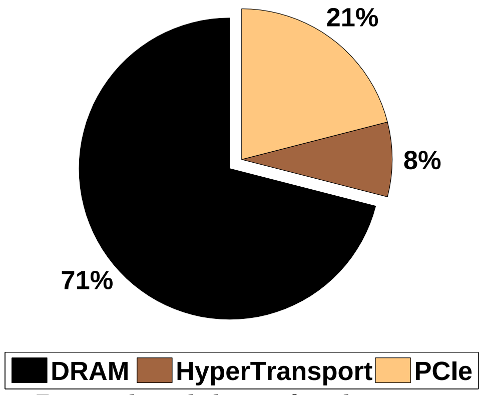


FIGURE 4.4: Power breakdown for data communication

sistent with prior measurements on physical machines. For the DRAM models, we use detailed timing and power parameters from manufacturer specifications [75, 91]. For the inter-processor models we use the HyperTransport Consortium's technology parameters [45] to estimate the latency attributed to packet processing and transmission. Our estimates align with NUMA measurements for pointer-chasing workloads [10, 71, 95]. Inter-processor power numbers are also consistent; we estimate 40W peak and 10-20W at typical utilization, which corroborates industry measurements of 12-16W [7]. Additionally, our inter-blade latency estimates are consistent with emulation parameters for disaggregated memory [60] and out inter-blade power estimates are derived from manufacturer specifications for PCIe bridges and link interfaces [56, 84].

We arrive at these estimates using parameterized models, which we design to accommodate a range of technology options . Similar methodology can be used to derive estimates for other technology generations or protocols. For example, we consider 2Gb DDR3 DRAMs for memory because they provide the requisite capacity for big memory servers. Our extensible models can accommodate other DRAM parts and emerging memory technologies (e.g., LPDDRx [68], fast sleep modes [69], and PCM [41]).

We consider HyperTransport 2.0 because it is a widely used and open protocol. Our models can be revised for HyperTransport 3.0 when its specifications become publicly available. Importantly, our models are equally relevant to Intel's QuickPath Interconnect, which relies on the same fundamental technology – packetized communication over fast, point-to-point links.

The current models focus on communication within a server and neglect inter-server networks. Although emerging 40GbE and 100GbE networks provide compelling bandwidth and may be relevant for key-value stores, they do not provide the low latency we require for direct load/store access to remote memory (at the time of this research). Ethernet latency is several orders of magnitude greater than remote memory access latency, which is at most a few hundred nanoseconds. Note that all the estimates in this project are based on technologies and white papers available around 2012, as the research is conducted between year 2012-2013.

4.3 Case Studies in Managing NUMA

We illustrate opportunities for parameterized design and management with two case studies that deploy our technology models. First, we adapt queueing policies to the costs of varied communication paths. Second, we compare the energy-efficiency of remote execution against migration.

4.3.1 Execution Model

Run-time systems rely on task queues to produce parallelism while preserving a programming models clean abstractions (e.g., MapReduce, Spark, GraphLab). The task at the head of the queue is likely to find its data already in main memory because of in-memory caching or software-level data prefetching. For example, Spark caches data from the current iteration of a machine learning kernel to ensure its availability for the next one. However, non-deterministic queueing complicates the coordination of task scheduling and data placement; the data placement mechanism cannot predict which core will become available at a certain time. A lack of coordination exposes NUMA in blade servers. Tasks must navigate multiple levels of NUMA and still guarantee service quality. We look beyond the bimodal locality classification (e.g., local versus remote) and examine distance to data when scheduling tasks with NUMA.

4.3.2 Task Queues and Refusal Scheduling

Refusing Execution. We draw inspiration from delay scheduling [115], which improves storage locality for MapReduce tasks. For an in-memory computational application, when a task arrives at the head of the task queue in a blade server, the next available core may not have its data prefetched in local memory. Instead, the data may have been prefetched into another processor or blade's memory. Our case studies present policies that dictate whether a queued task should refuse execution on an available core due to NUMA.

Figure 2 illustrates refusal scheduling. Arriving tasks enter a queue. When a core becomes available, the scheduler determines this cores proximity to data required by the next task (e.g., address of tasks vertex given an graph in distributed memory). The schedulers refusal policy selectively refuses and permits NUMA. Refusing tasks enter a high-priority retrial queue. Queues prioritize tasks with earlier arrival times

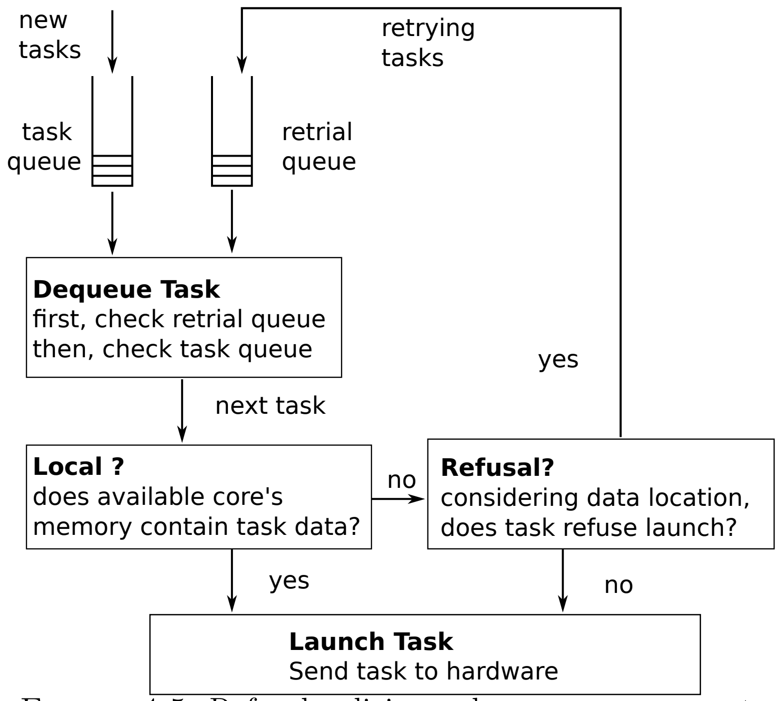


FIGURE 4.5: Refusal policies and queue management.

for FIFO fairness. A refusal limit is used to avoid starvation. We consider four refusal policies for multiple NUMA levels (e.g., Figure 4.5).

Refusal Policies. We demonstrate refusal policies for the blade server illustrated in Figure 4.2, which supports three types of remote access: inter-processor with one hop, inter-processor with two hops, and inter-blade. We define refusal polices according to the degree a task tolerates remote data access.

- **Local Execution (Local).** Task accepts execution on processor for which its data is local. Otherwise, task refuses.

- **Inter-processor 1-Hop Execution (IP-1).** Task prefers execution on processor for which its data is local. It also accepts execution on processor that is one inter-processor link away. Otherwise, task refuses.

- **Inter-processor 2-Hop Execution (IP-2).** Task prefers execution on blade for which its data is local. The task refuses execution if its data resides on another blade.

- **Inter-blade Execution (IB).** Task prefers execution on a processor for which its data is local. It then prefers execution on a blade for which its data is available via inter-processor links. Otherwise, task executes on any available core via inter-blade links.

The policies are presented in order of increasing NUMA tolerance. Local is least tolerant; tasks require local memory and refuse even one hop to another processor's memory. In contrast, IB tolerates both inter-processor and inter-blade communication.

Multi-Queue Implementations. Although Figure 4.5 shows a single queue, refusal scheduling generalizes to multiple queues with round-robin load balancing. Instead of one queue for the server, suppose we implement one queue per processor. When a core becomes available, the task at the head of each queue applies its refusal policy. Among the tasks that do not refuse, the oldest executes. This policy for m queues approximates a single-queue policy that relaxes FIFO and checks the next m tasks when a core becomes available. Multiple queues and the single queue provide the same mean queueing time, but queueing time variance increases with multiple queues. This case study can be generalized to multiple queue implementations with modification to the simulation environment.

4.4 Multi-scale Simulation

We present a multi-scale framework that links detailed processor and memory simulators to a discrete event queueing simulator, thereby permitting the first coordinated study of NUMA and queueing dynamics. Memory simulation quantifies NUMAs effect on average memory access time, processor simulation quantifies the effect on instruction throughput, and queueing simulation quantifies the effect on system throughput.

4.4.1 Methodologies

Processor and Memory Simulator. We integrate the Marssx86 full-system processor simulator [83] with the DRAMSim2 memory simulator [91]. Accessing local memory requires 50ns in addition to any queueing delays at the memory controller and any buffer/bank conflicts at the DRAMs. We further add inter-processor and inter-blade latencies, from the technology model, when accessing remote memory. We parameterize latency to assess performance penalties when communicating via one HyperTransport hop, two HyperTransport hops, and PCIe.

Queueing Simulator. We implement a discrete event simulator, modeled after BigHouse [72, 73], to capture datacenter queueing dynamics. BigHouse uses service time from physical measurements, which is realistic but inflexible. In contrast, we embed architectural parameters (derived in this section) into cycle-accurate processor and memory simulators to estimate service time. This estimate accounts for NUMA delays, which vary according to the length of the communication path from core to memory. Service time dictates response times in a G/G/k queue,[1] which cannot be captured in closed-form equations and must be simulated. The queueing simulator tracks each task's response time and reports summary statistics for datacenter service service (e.g., 95th percentile).

Energy cost depends on the amount of data transferred and the links used. Processor simulation reports the amount of data transferred by remote accesses. Queueing simulation reports the number of tasks that use each link. We aggregate these statistics to estimate the average communication power consumed in the system. And we exploit our models and simulators to evaluate new management strategies

[1] Generalized inter-arrival time, generalized service time, k servers.

#	Name	Bandwidth (Intensity, MB/s)		Penalty HT-1	Penalty PCIe
Apache Spark					
S1	Word count	MI	264	1.17x	1.67x
S2	Logistic regression	MI	242	1.17x	1.60x
S3	Pagerank	MI	276	1.22x	1.76x
S4	Transitive closure	MI	235	1.19x	1.62x
S5	Alternating least squares	MI	231	1.16x	1.55x
S6	K-means	MI	283	1.22x	1.76x
S7	Pi	CI	21	1.01x	1.04x
Phoenix MapReduce					
M8	Word count	MI	111	1.10x	1.33x
M9	Histogram	CI	42	1.02x	1.05x
M10	String match	CI	28	1.04x	1.12x
M11	Linear regression	CI	7	1.0x	1.0x
Parsec					
P12	Facesim	EMI	2159	1.39x	2.46x
P13	Dedup	CI	50	1.09x	1.19x
P14	Bodytrack	CI	35	1.02x	1.06x
P15	Vips	CI	23	1.01x	1.04x
P16	Ferret	CI	73	1.04x	1.18x
P17	Raytrace	CI	19	1.02x	1.06x
P18	Swaptions	CI	5	1.00x	1.00x
P19	Streamclusters	CI	4	1.00x	1.00x
P20	Blackscholes	CI	3	1.00x	1.00x

Table 4.2: Benchmarks, their demand for remote memory bandwidth (Extremely Memory-Intensive, Memory Intensive, Compute Intensive) and latency penalties from NUMA.

for blade servers.

4.4.2 Benchmarks and Parameters

Benchmarks. We evaluate twenty benchmarks drawn from Apache Spark [116], Phoenix MapReduce [87], and Parsec [17]. These workloads operate on big data, demand throughput, and represent the class of applications targeted by blade servers. Spark addresses performance limitations in MapReduce for iterative machine learning algorithms. MapReduce performs I/O to the distributed file system between the map and reduce phases, which is inefficient within an iterative algorithm such as

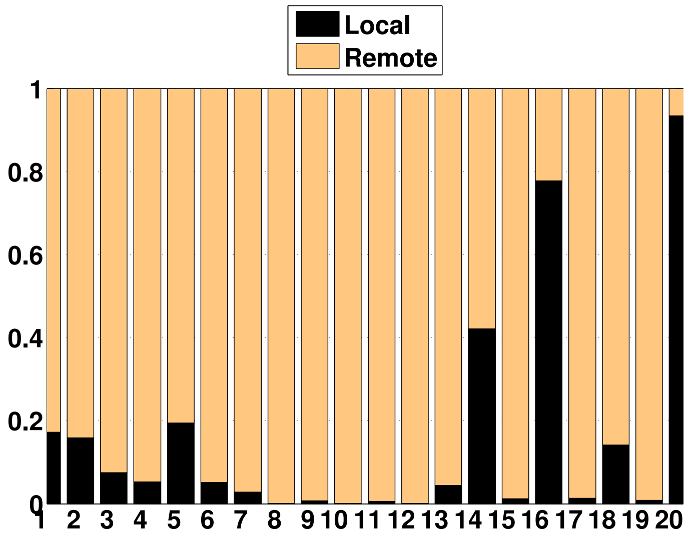


Figure 4.6: Local vs Remote Memory Access.

K-means. In contrast, Spark keeps data in memory across computational iterations to improve performance by an order of magnitude. We evaluate both Spark and Phoenix, an implementation of MapReduce for shared memory machines. Finally, Parsec workloads benchmark data analytics.

Table 4.2 classifies applications according to memory intensity. We measure bandwidth demand and classify applications as (extremely) memory-intensive or compute-intensive. Bandwidth demand correlates with NUMA-induced performance penalties. Extremely memory-intensive tasks must execute on cores with data in local memory; these tasks would otherwise saturate interconnect bandwidth. The remaining workloads are amenable to a mix of local and remote execution.

Local and Remote Access. Figure 4.6 shows the percentage of local and remote data accesses. We focus our evaluation on remote accesses to prefetched data and assume that local memory holds code and stack contents. NUMA tolerance varies widely, even among memory-intensive applications, because tasks access memory at different rates and not all accesses address the heap. The simulator identifies address space regions for code, stack, and heap, using region boundaries to differentiate

Core	4-way OoO core (2GHz)
L1 Cache	128KB Private L1 Instruction cache 128KB Private L1 Data cache
L2 Cache	2M L2 8-way associate write-back, 64B cache line size
DRAM	DDR3, clock 667MHz, 50ns
Remote Comm. Latency	One HyperTransport hop: 100 ns Two HyperTransport hops: 130 ns PCIe: 360 ns

Table 4.3: Marssx86 and DRAMSim2 simulation parameters.

remote memory requests from local ones. We track statistics for user mode and neglect those in kernel mode.

Simulation Parameters. Table 4.3 summarizes processor and memory simulation parameters. We study the big memory blade server of §4.2, which represents current server design [29, 48]. We implement four scheduling polices – **Local**, **IP-1**, **IP-2** and **IB** – that selectively permit and avoid NUMA. Each policy permits a task to refuse execution up to the refusal limit. A small refusal limit (1-10) is insufficient in a many-core server. On the other hand, a large refusal limit (1000) significantly delays execution. To strike a balance, we limit the maximum number of refusals to 100.

4.5 Evaluation

We use our models and simulators to evaluate case studies in NUMA mitigation. We find that the optimal refusal pol- icy varies according to a tasks memory intensity and system utilization. In addition, we discover that permitting fine- grained NUMA is more energy-efficient than coarse-grained page migration.

4.5.1 Throughput

Avoiding NUMA reduces task service time and increases throughput. A throughput analysis favors restrictive policies, which refuse execution on cores that require remote

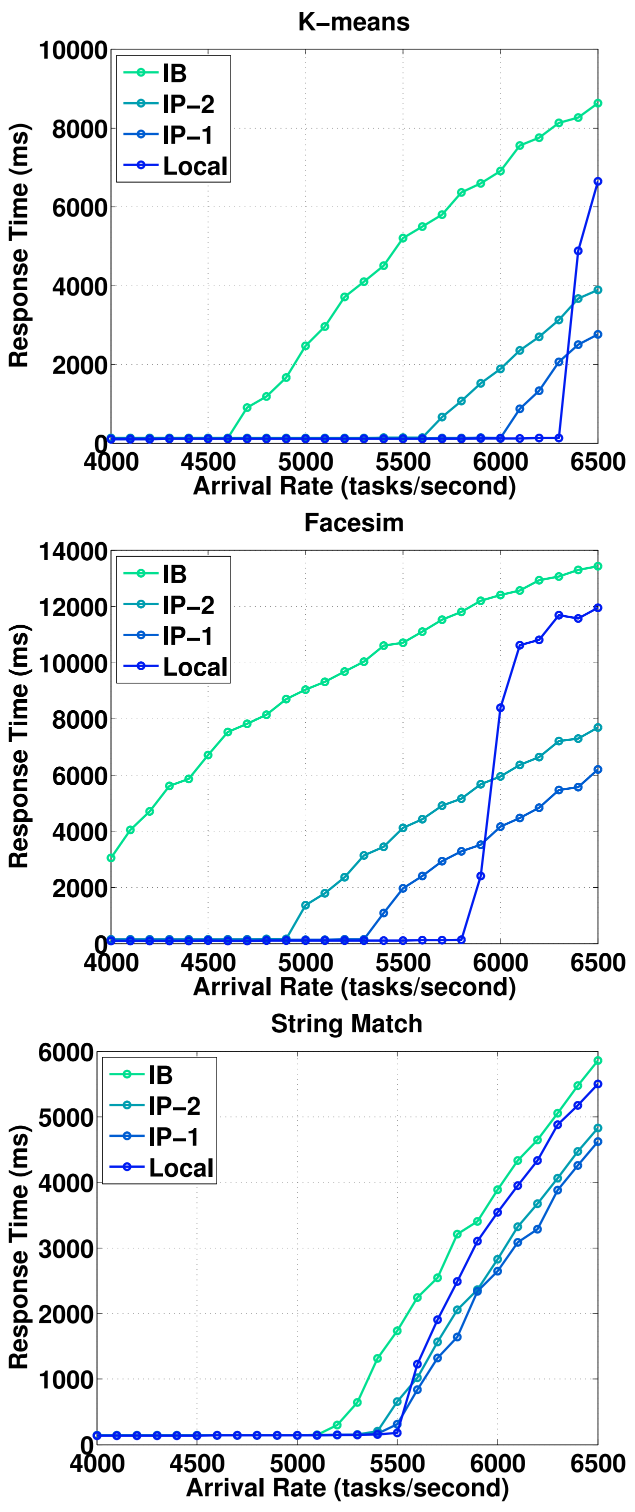


FIGURE 4.7: Throughput as a function of task arrival rate for representative tasks drawn from (a) Spark K-means, (b) Parsec facesim, and (c) MapReduce string match.

data access. For example, **Local** and **IP-2** policies restrict data movement and limit service time degradation. We detail the relationship between refusal scheduling and throughput in two steps. First, we quantify service time for an individual task. Then, we quantify the maximum sustainable throughput for a stream of tasks within a blade server.

We simulate an individual task and quantify its performance under three scenarios for data movement – using local memory, using inter-processor communication, and using inter-blade communication. We determine how NUMA penalizes instruction throughput and extend these penalties to task service rate. For example, suppose each task executes 100M instructions, core frequency is 2.5GHz, and core IPC is 1. Each core services 25 tasks per second and a blade server with 256 cores services 6,400 tasks per second. This analysis assumes each task uses local memory. In practice, however, some tasks use remote memory and NUMA penalties degrade throughput.

Figure 4.7 shows how avoiding NUMA increases throughput. Let us define a server's maximum sustainable throughput $\lambda_{\mathbf{max}}$ as the highest task arrival rate the machine can accommodate without increasing response times. As tasks arrive more quickly and the rate exceeds $\lambda_{\mathbf{max}}$, tasks accumulate in the queue and response time increases. A server can sustain higher arrival rates by refusing NUMA execution, thereby reducing service time and increasing service rate. **Local** maximizes throughput while **IB** penalizes it.

The x-intercept in each figure identifies $\lambda_{\mathbf{max}}$. For Spark K-means, the server sustains only 4.6K tasks per second under **IB**, which permits all communication and NUMA execution, and sustains 6.4K tasks per second under **Local**, which refuses all communication. We observe similar trends for all memory-intensive tasks. But compute-intensive tasks, such as MapReduce StringMatch, are insensitive to NUMA and $\lambda_{\mathbf{max}}$ is insensitive to refusal policy.

Figure 4.8 summarizes our throughput analysis for the benchmark suite, showing

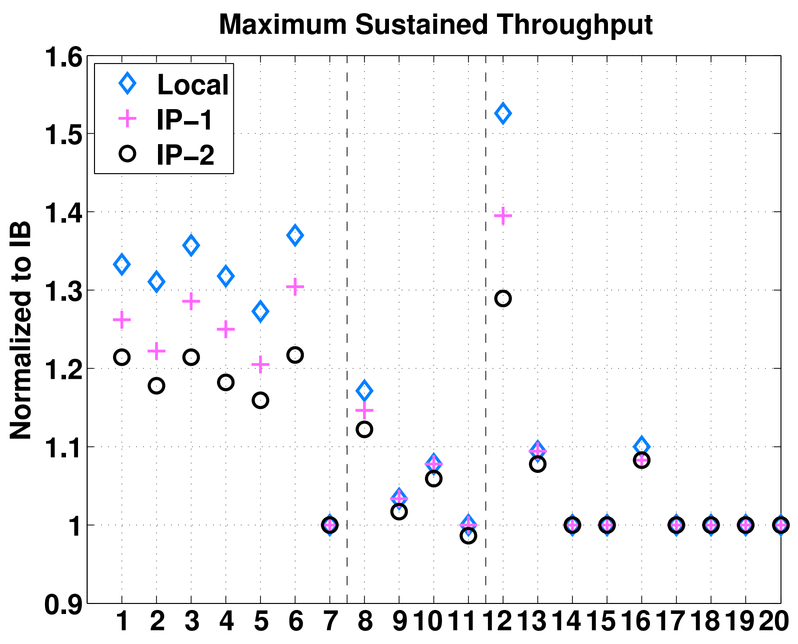


FIGURE 4.8: Maximum sustained throughput for varied benchmarks indexed in Table 4.2. Vertical dash lines separate benchmark groups: S1-S7, M8-M11, P12-P20. This figure format will be consistent for all the graphs in this section.

how throughput improves as the refusal policy becomes more restrictive. With a policy that favors local execution, tasks encounter lower service time and higher service rates. And this effect is most pronounced for memory-intensive workloads (S1-S6, M8, P12), which suffer more from NUMA penalties. Ranking the policies by increasing throughput gives: **IB**, **IP-2**, **IP-1**, **Local**. Note, however, that aggregate throughput is only one measure of system performance. When we consider individual task latency, judiciously permitting NUMA proves beneficial.

4.5.2 Response Time

Judiciously permitting NUMA reduces queueing delay and response time for individual tasks. Dequeueing a task when a core becomes available means less time in the arrival and retrial queues. A latency analysis favors permissive policies, which allow execution on cores that require remote data access. However, latency is not determined by NUMA policy alone. Policy interacts with system utilization and a task's NUMA sensitivity. We simulate two scenarios – highly and lightly loaded servers – and analyze queueing dynamics for millions of tasks. We calculate the 95th percentile for response time; recall that response time is queueing time plus service time.

Figure 4.9 reports the 95th percentile for response time. NUMA execution can be beneficial, even when tasks are memory-intensive (S1-S6, M8, P10) and server load is high. As seen in the throughput analysis, refusing NUMA (e.g., **Local**) reduces service time by improving locality. But refusing NUMA also increases queueing time by causing tasks to re-enter the queue for scheduling. If every tasks refuses NUMA, queueing delays accumulate and harm end-to-end response time. Thus, judiciously permitting NUMA is beneficial.

NUMA execution can be beneficial, even when tasks are memory-intensive (S1-S6, M8, P10) and server load is high. **IP-1** and **IP-2**, which permit inter-processor communication with one and two hops, consistently perform better than **Local**. We explain this effect by examining a task's likelihood of de-queueing when a core becomes available. If a task adopts **Local** and refuses communication, the task de-queues to execute on one specific processor, the one with its data in local memory. However, if a task adopts **IP-1** and tolerates one communication hop, the task de-queues to execute on any one of three processors, the one with its data and the two that are one-hop away in the mesh. In other words, a task that adopts **IP-1** instead of **Local** is 3× more likely to exit the queue when a core becomes available. More choices shorten queueing times, which shorten response times.

Dequeueing quickly is even more attractive for compute-intensive tasks (S7, M9, M11, P13-P20). These tasks require little data movement. Refusing NUMA execution to avoid communication does not reduce service time and only increases queueing time. Compute-intensive tasks can adopt **IB**, which exploits the PCIe backplane for direct load/stores to another blade's memory, without penalty. Indeed, refus-

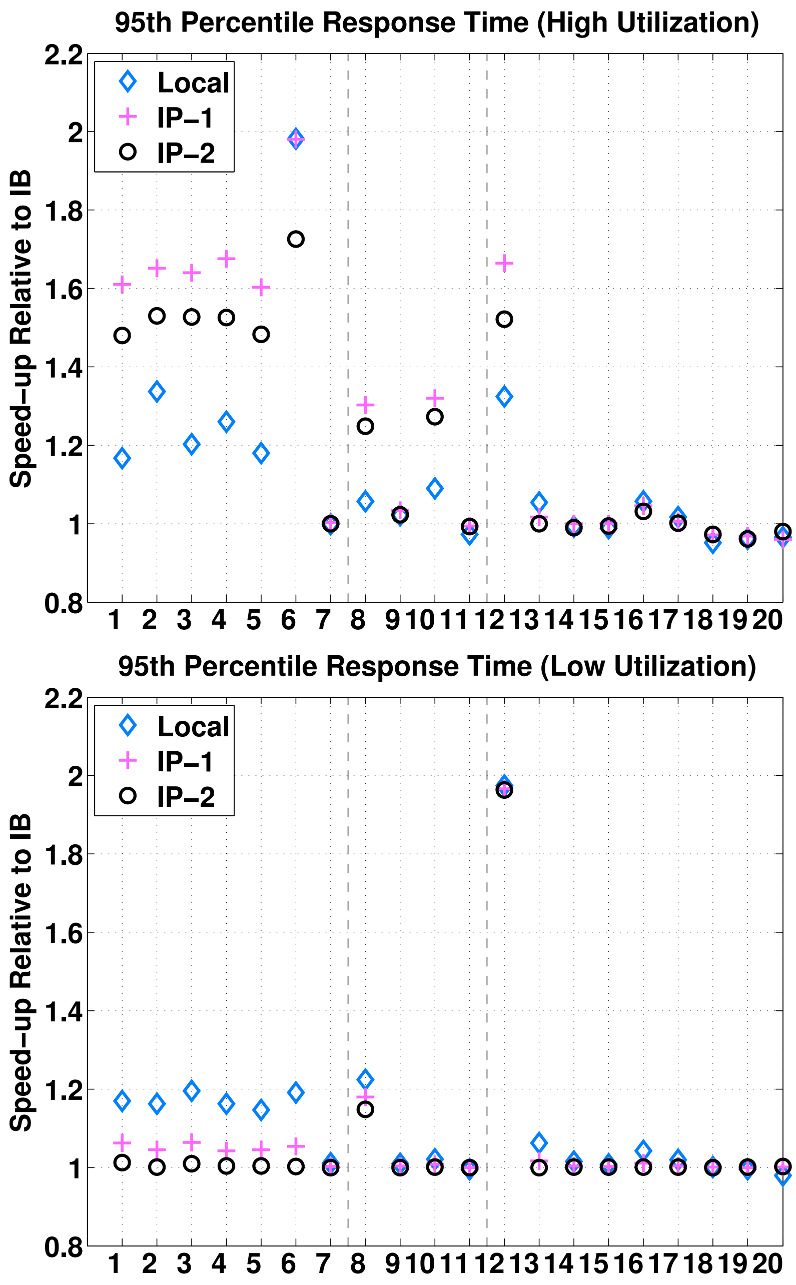


FIGURE 4.9: 95th percentile response time. Response times normalized to IB policy, which permits all NUMA accesses. Data shown for (a) high server utilization and (b) low server utilization.

ing remote execution by adopting **Local** can harm response time as queueing time increases – see tasks P18-P20.

Finally, suppose the server is lightly loaded. Because many cores are available, queueing time is less important and service time dominates response time. In this setting, refusing NUMA execution and safeguarding service time performs better but not by much. **Local** out-performs less restrictive policies like **IB** and **IP-2** by less than 20% for memory-intensive tasks and by much less for compute-intensive ones. From a different perspective, this data favors judiciously permitting NUMA because doing so penalizes performance by less than 20%.

Summary. Our models and analysis produces three conclusions for adapting refusal policies to different system settings with model hardware:

1. Compute-intensive tasks should permit NUMA (**IB**) since tasks are insensitive to memory latency.
2. Memory-intensive tasks should selectively permit NUMA (**IP-1**, **IP-2**) in highly loaded servers to balance service time and queueing time.
3. Memory-intensive tasks should avoid NUMA (**Local**) in lightly loaded servers to reduce service time since queueing time is negligible.

4.5.3 Comparison to Data Migration

Task scheduling policies that permit NUMA exploit fine-grained load/store access to remote memory. Remote access retrieves the data needed to fill a cache line instead of migrating entire pages from remote locations into local ones. Although migration is fast, its energy costs are high. We compare the energy costs of remote access and data migration. Our results show that migration energy is 1.5-5.0x greater than that of remote access for memory-intensive workloads.

Energy cost depends on the amount of data transferred and the links used. The processor and queueing simulations report data transferred by remote accesses and tasks that use each link. First, to estimate the amount of data migrated, we count the number of unique pages accessed by a program during a 100M-instruction simulation and multiply by the page size (4KB). After a page has migrated into local memory, the processor core retrieves the desired cache line. We use D_{page} to denote the amount of data moved during page migration and use D_{access} to denote the amount of data actually accessed to fill a cache line.

$$E_{\text{Migrate}} = D_{\text{page}}(2E_{\text{Mem@100}} + E_{\text{Link}}) + D_{\text{access}}E_{\text{Mem@20}} \tag{4.1}$$

Equation (4.1) estimates the page migration energy. Page migrations access DRAM twice – once to read from remote memory and once to write into local memory – and transfer data at high bandwidth. In addition, these memory transfers may traverse inter-processor and inter-blade links. We estimate the energy costs for both types of communication. DRAM cost when transferring data at full bandwidth is $E_{\text{Mem@100}}$ =70pJ/bit. Link cost, based on the expected number of inter-processor hops, is E_{Link}=110pJ/bit – see §4.2. Finally, filling cache lines incur a relatively high DRAM cost because these accesses do not fully utilize channel bandwidth, which increases the average cost to $E_{\text{Mem@20}}$ =110pJ/bit.

$$E_{\text{Remote}} = D_{\text{access}}(E_{\text{Link}} + E_{\text{Mem@20}}) \tag{4.2}$$

For comparison, Equation (4.2) estimates the cost of direct access to remote memory. The simulator counts last-level cache misses, ignoring those for local code and stack, and multiplies by the cache line size (64B) to estimate the amount of data accessed remotely (D_{access}). Remote access incurs costs for links and for DRAM at the remote location.

To note that, the amount of input data that needs to be copied is estimated by the unique number of pages that have been touched by the program during the execution. To make this characterization more accurate, simulating the entire program is needed to model the cache misses across all the application phases. With those data, we can use the actual input data size instead of the page size. Due to the long simulation time of all the applications, we only profiled 100M instructions on the region of interest.

Energy Efficiency. Figure 4.10 shows that the energy cost of data migration is 1.5-5.0$\times$ that of remote access. Remote access is less expensive because it transfers fine-grained cache lines, not coarse-grained memory pages. However, we might expect migration to perform better if a task transfers a page to access it multiple times, thereby amortizing the cost of reading a page from remote memory and writing it into local memory. We determine the number of page accesses that are required to make migration competitive with remote access. This break-even point, where Equations 4.1 and 4.2 are equal, is shown below for several communication paths.

- Inter-blade: $D_{\text{access}}/D_{\text{page}} = 2.2\times$
- Inter-processor two-hop: $D_{\text{access}}/D_{\text{page}} = 2.9\times$
- Inter-processor one-hop: $D_{\text{access}}/D_{\text{page}} = 4.8\times$

Migration is less expensive than remote access to another blade if each byte in the migrated page is accessed 2.2$\times$, on average. Likewise, migration is less expensive than remote access to another processor if data in the migrated page is used more than 2.9-4.8$\times$. Migration's break-even point given inter-processor communication is higher because remote access across HyperTransport or QuickPath interconnects consumes little energy.

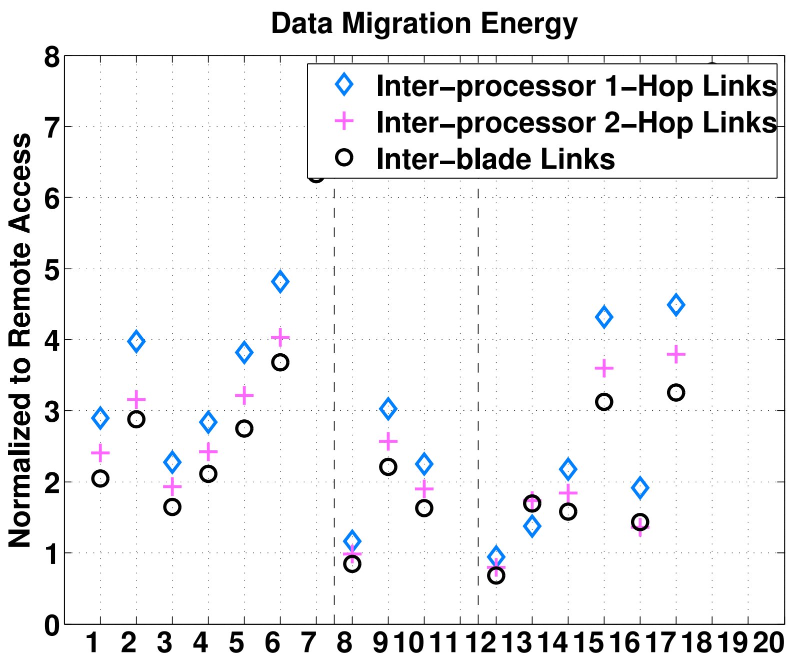


FIGURE 4.10: Data migration versus remote access (Energy). Benchmarks 18-20 are out of scope; baseline energy is too small for compute-intensive workloads with little memory activity.

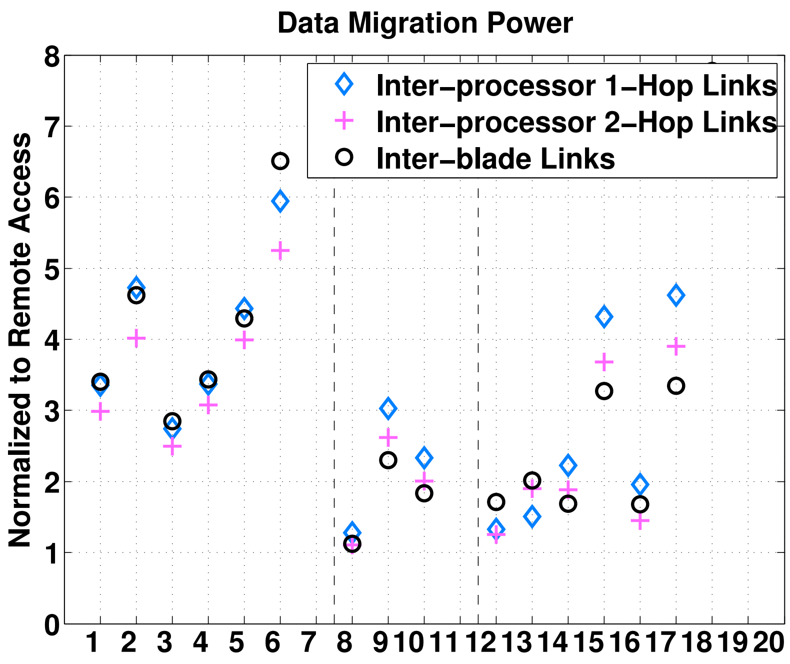


FIGURE 4.11: Data migration versus remote access (Power)

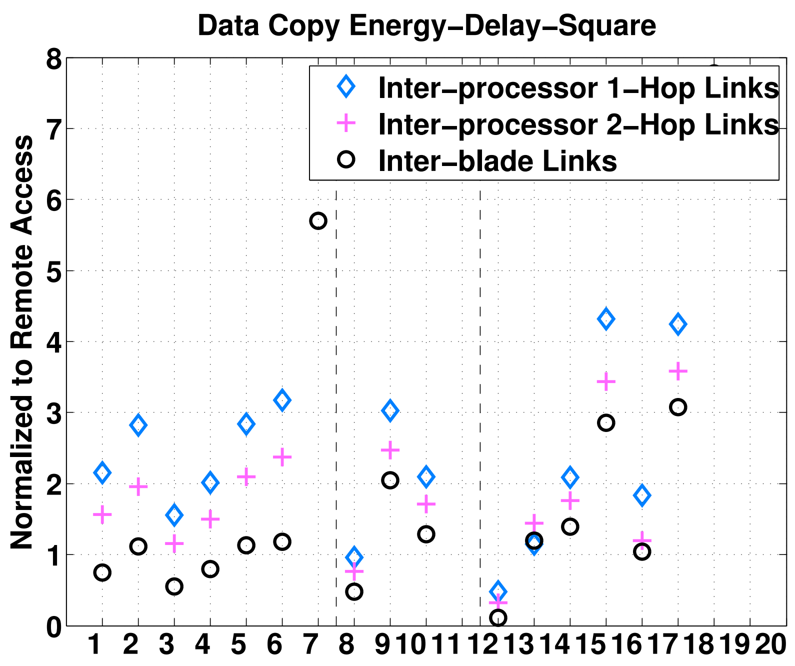


FIGURE 4.12: Data migration versus remote access (ED^2).

Figure 4.11 shows that migration dissipates more power than remote access. Moreover, it shows that migration is an inefficient mechanism to improve performance. For example, inter-blade remote accesses increase task delay by 1.6-1.7× – see Table 4.2. Migration eliminates this slowdown by avoiding remote accesses but increases communication energy by 2.0-8.0× to do so – see Figure 4.10. Such asymmetric energy-delay trade-offs encourage policies that selectively permit NUMA instead of data migration.

Finally, Figure 4.12 presents the ED^2 metric for energy efficiency. This metric is equivalent to performance-cubed per Watt, which is a voltage invariant metric derived from the cubic relationship between power and voltage [21]. Moreover, this metric emphasizes performance over power. In this setting, data migration is preferred when remote access requires inter-blade communication, which more heavily penalizes performance. But remote access is still preferred when remote access requires inter-processor communication.

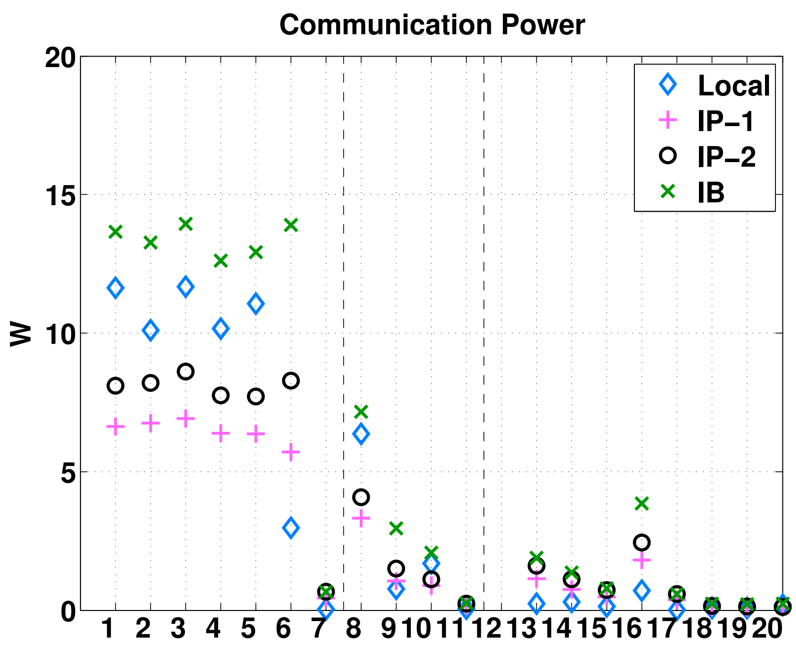


FIGURE 4.13: Communication power for highly utilized server.

4.5.4 *Power Efficiency Analysis*

Figure 4.13 presents the average communication power for four degrees of NUMA refusal. Refusal policies divert data flow between various paths and interconnects. Some of these paths dissipate more power than others. Intelligently refusing execution reduces power by eliminating communication with distant memory. Ranking the policies by increasing power efficiency gives: **IB**, **Local**, **IP-2**, **IP-1**. This ranking is determined by how each refusal policy distributes data movement across links in the system.

Figure 4.14 shows the distribution of task data accesses for the NUMA-refusal policies. To understand this graph and its implications for power, consider the **Local** scheduler, which refuses execution on available cores unless task data resides in local memory. The scheduler limits the maximum number of refusals. Once a task reaches this limit, it must dispatch to the next available core. Most tasks execute locally but tasks that refuse too often risk inter-blade communication. This risk means that

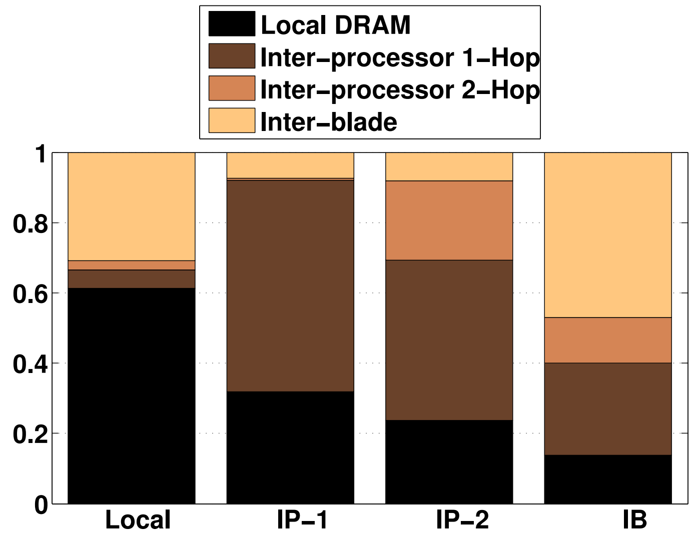


FIGURE 4.14: Activity distribution across channels (stacked bars) for tasks following one of four refusal policies (x-axis).

Local, dissipates more power than **IP-1** and **IP-2** policies, which selectively permit inter-processor communication. Permitting all types of remote data access with **IB** is obviously power-intensive.

Figure 4.13 presents power numbers that are consistent with reasonable link utilization. For example, the memory-intensive benchmarks S1-S6 running with the **IB** scheduler dissipates 13W. From this number, we can infer link utilization. HyperTransport and PCIe dissipate 40W and 60W at peak utilization, but we expect 60% and 40% utilization on these communication paths given the distribution in Figure 4.14 and utilized power scales to 48W. The fact that the simulator reports 13W for the memory-intensive S1-S6 suggests that these Spark workloads use about 25% of the link bandwidth.

In summary, we find that remote access and NUMA execution is more efficient than data migration. Moreover, NUMA-aware scheduling policies reduce power and energy by shifting communication towards more efficient paths. Finally, the power analysis indicates that NUMA execution produces moderate link utilization throughout the blade server.

4.6 Conclusions and Future Work

In this chapter, we analyze communication costs in blade servers, deriving technology models for system simulation. We present case studies that coordinate NUMA mitigation and queue management. These studies illustrate benefits in throughput, response time, and energy efficiency for various applications. Our results motivate further hardware design space exploration and sophisticated task management for in-memory computing on blade servers.

Our modeling methods lay the foundation for coordinated design of software scheduling policies and hardware interconnection topologies. The current framework allows the system scheduler (NUMA-aware scheduler) to choose a tolerable NUMA level for applications targeting performance or energy goals. The optimal refusal scheduling policy changes when the optimization goal switches from response time to energy-efficiency. However, a centralized scheduler faces challenges adjusting for heterogeneous environments; i.e., heterogeneous tasks, diverse user preferences, system goals and data distribution. For future studies, such memory locality-aware scheduling problem resembles the pricing problem for city parking [25]; microeconomic models can be an effective approach to decentralize the scheduling process and satisfy diverse users' preferences. Such a formalism may be an attractive future work. Decentralized updating rules define prices of each memory location based on supply, demand, data hot spot and nearby interconnection traffic. Then each task acts on behalf of itself to choose the location, pays the price and occupies the memory location. A preliminary study of this extension is done during the author's PhD study. This thesis does not include details of this study.

4.7 Acknowledgment

This work is partially supported by NSF grants CCF-1149252 (CAREER), CCF-1337215 (XPS-CLCCA), and AF-1408784. This work is also partially supported by STARnet, a Semiconductor Research Corporation Program, sponsored by MARCO and DARPA. Any opinions, findings, or conclusions expressed in this material are those of the author(s) and do not necessarily reflect the views of these sponsors.

5

Conclusion

This thesis proposes to use microeconomic models for managing shared datacenters and presents three pieces of work to demonstrate the workflow and effectiveness of this approach. In each work, we design system management frameworks that can improve server utilization and system performance while guaranteeing shared user fairness. The definition of fairness differs when the system setting changes. For instance, in the first work, our definition for fair task colocation is stability, satisfied preferences and fair attribution of colocation costs; in the second work, we pursue fair resource allocation that guarantees sharing incentives, envy-freeness and Pareto-efficiency. Both fairness properties are defined from a game-theoretic perspective to incentivize strategic users to share. With related microeconomic models, we are able to formalize the problem, develop management mechanisms that can provide competitive system performance while guaranteeing game-theoretic desiderata.

Our contribution is not limited to adapting microeconomic models for system problems. Being able to provide satisfactory system performance, we must develop accurate performance models that can quickly capture applications' behavior to hardware in datacenters. The coordinated design approach of lightweight performance models that can drive the system management mechanisms is a bigger contribution. In this thesis, we experiment with three types of performance modeling techniques, ranging from fitting utility functions, to training machine learning models from historical data, to developing our own parametric models from system simulations. Each method has its own strength and challenges.

Utility functions are the simplest models which provide a concise interface for designing the management mechanisms. However, these functions can hardly capture diverse types of applications' behavior and often times one model is needed for each type of application. Machine learning models such as classification techniques can overcome these difficulties in performance modeling. The downside is that there are limited management algorithms that can use the output of these models for making resource allocation or scheduling decisions. Finally, the most complicated performance model used in this thesis is system, architectural simulation combined with derived technology models. One use case for such a detailed model is to reduce the design costs and cycles for configurable servers and to provide performance and energy estimates for various hardware configurations and technology selections. The other use case for this model is to analyze the application performance and energy effect under different system management policies for a fixed server configuration.

The results presented in the dissertation, either with simulations or real system measurements, are just the beginning of this type of research. Each work can be improved to various degrees and to different directions. Details are stated in the conclusions and future work section under the corresponding chapter. During the presentation of this research, I always get one important question "Why do we care about fairness in systems?" The answer is obvious from the perspective of human behavior but subtle from the perspective of systems management. Why do company management systems need fairness? Why do voting systems need fairness? Because in any system without fairness, human beings as selfish animals will greedily pursue their goals until resources are all consumed. There is no guarantee that every human being will have an equal opportunity to benefit from these resources. And in order to give an equal opportunity to benefit from these resources, fairness is a common goal in many management policies. As human users are behind every application, guaranteeing application's fairness is similar as giving each user a fair chance to "survive". Therefore, it should be considered as an important system outcome. The challenge still lies in the design of management policies that are both fair and efficient.

Bibliography

[1] "Akka scala documentation," http://doc.akka.io/docs/akka/2.4/AkkaScala.pdf.

[2] "Docker," http:/docs.docker.com.

[3] "Movielens," http://grouplens.org/datasets/movielens/.

[4] "Us census data (1990) data set," https://archive.ics.uci.edu/ml/datasets/US+Census+Data+(1990).

[5] "Web data commons: Hyperlink graphs," http://webdatacommons.org/hyperlinkgraph/index.html.

[6] D. J. Abraham, P. Biró, and D. F. Manlove, "Almost stable matchings in the roommates problem," in *Approximation and online algorithms*. Springer, 2006, pp. 1–14.

[7] AMD, "The truth about power consumption starts here," in *AMD White Paper: Power Consumption*, 2009.

[8] G. Amdahl, "Validity of the single processor approach to achieving large-scale computing capabilities," in *Proceedings AFIPS Conference*, 1967.

[9] J. Ang, R. Ballance, L. Fisk, J. Johnston, and K.Pedretti., "Red Storm capablity computing queueing policy," in *Cray Users' Group (CUG)*, 2005.

[10] J. Antony, P. Janes, and A. Rendell, "Exploring thread nad memory placement on NUMA architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport," in *HiPC*, 2006.

[11] Aristotle, *Nicomachean Ethics*.

[12] E. Arkin, S. Bae, A. Efrat, K. Okamoto, J. Mitchell, and V. Polishchuk, "Geometric stable roommates," *Information Processing Letters*, vol. 109, no. 4, pp. 219–224, 2009.

[13] K. Arrow and G. Debreu, "Existence of an equilibrium for a competitive economy," *Econometrica: Journal of the Econometric Society*, 1954.

[14] L. Barroso and U. Hoelzle, *The datacenter as a computer: An introduction to the design of warehouse-scale machines.* Synthesis Lectures on Computer Architecture, 2009.

[15] L. A. Barroso, J. Clidaras, and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines," *Synthesis lectures on computer architecture*, vol. 8, pp. 1–154, 2013.

[16] O. A. Ben-Yehuda, E. Posener, M. Ben-Yehuda, A. Schuster, and A. Mu'alem, "Ginseng: Market-driven memory allocation," in *Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments.* ACM, 2014.

[17] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *The International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[18] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali, "A case for NUMA-aware contention management on multicore systems," in *PACT*, 2010.

[19] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou, "Apollo: Scalable and coordinated scheduling for cloud-scale computing." in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[20] S. Boyd and L. Vandenberghe, *Convex optimization.* Cambridge university press, 2004.

[21] D. Brooks *et al.*, "Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors," *IEEE Micro*, vol. 20, no. 6, 2000.

[22] E. Budish, "The combinatorial assignment problem: Approximate competitive equilibrium from equal incomes," 2011.

[23] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica, "HUG: Multi-resource fairness for correlated and elastic demands," in *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[24] S. Clearwater and S. Kleban, "ASCI Queueing Systems: Overview and comparisons," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2002.

[25] I. R. Cohen, A. Eden, A. Fiat, and Ł. Jeż, "Pricing online decisions: Beyond auctions," in *Proceedings of the 26th annual ACM-SIAM symposium on discrete algorithms (SIAM)*, 2014.

[26] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth, "Traffic management: A holistic approach to memory placement on NUMA systems," in *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[27] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[28] ——, "Quasar: resource-efficient and qos-aware cluster management," in *Proceedings of the 19th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[29] Dell, "Dell PowerEdge M820 Blade Server," 2014. [Online]. Available: http://www.dell.com/us/business/p/poweredge-m820/pd

[30] M. Dong, T. Lan, and L. Zhong, "Rethink energy accounting with cooperative game theory," in *Proceedings of the 20th annual international conference on Mobile computing and networking (MobiCom)*. ACM, 2014, pp. 531–542.

[31] E. Ebrahimi, C. Lee, O. Mutlu, and Y. Patt, "Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems," in *Proceedings of the 15th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[32] S. Fan, S. Zahedi, and B. Lee, "The computational sprinting game," in *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[33] A. Fedorova, M. Seltzer, and M. Smith, "Improving performance isolation on chip multiprocessors via an operating system scheduler," in *Proceedings International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2007.

[34] J. Feigenbaum, C. Papadimitriou, and S. Shenker, "Sharing the cost of mulitcast transmissions (preliminary version)," in *Proceedings of the thirty-second annual ACM symposium on Theory of computing*. ACM, 2000, pp. 218–227.

[35] D. Gale and L. Shapley, "College admissions and the stability of marriage," *American Mathematical Monthly*, vol. 69, 1962.

[36] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types." in

Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2011.

[37] B. Goglin and N. Furmento, "Enabling high-performance memory migration for multithreaded applications on LINUX."

[38] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," in *Proceedings of the Annual Conference of the Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2014.

[39] M. Guevara, B. Lubin, and B. Lee, "Navigating heterogeneous processors with market mechanisms," in *Proceedings of the 19th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[40] D. Gusfield and R. W. Irving, *The stable marriage problem: structure and algorithms*. MIT press Cambridge, 1989, vol. 54.

[41] T. Ham, B. Chelepalli, N. Xue, and B. Lee, "Disintegrated control for power-efficient and heterogeneous memory systems," in *Proceedings of the 19th IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2013.

[42] Z. Han, C. Pandana, and K. Liu, "A self-learning repeated game framework for optimizing packet forwarding networks," in *Wireless Communications and Networking Conference, 2005 IEEE*, vol. 4. IEEE, 2005, pp. 2131–2136.

[43] M. Hill and M. Marty, "Amdahl's law in the multicore era," *Computer Society*, vol. 41, no. 7, 2008.

[44] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[45] B. Holden, "Latency comparison between HyperTransport and PCI-Express in communication systems," in *HyperTransport Consortium Technical Note*, 2006.

[46] R. Hou, T. Jiang, L. Zhang, P. Qi, J. Dong, H. Wang, X. Gu, and S. Zhang, "Cost effective data center servers," in *Proceedings of the 19th IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2013.

[47] HyperTransport Consortium, "HyperTransport link specifications," 2009. [Online]. Available: http://www.hypertransport.org/default.cfm?page=HyperTransportSpecifications

[48] IBM, "IBM System x3750 M4," 2014.

[49] Intel, "An introduction to the intel quickpath interconnect," in *Intel White Paper*, 2009.

[50] ——, "Intel Xeon processor C5500/C3500 series non-transparent bridge," in *Intel Doc 323328-001*, 2010.

[51] R. W. Irving, "An efficient algorithm for the stable roommates problem," *Journal of Algorithms*, vol. 6, no. 4, pp. 577–595, 1985.

[52] K. Iwama and S. Miyazaki, "A survey of the stable marriage problem and its variants," in *Informatics Education and Research for Knowledge-Circulating Society, 2008. ICKS 2008. International Conference on.* IEEE, 2008, pp. 131–136.

[53] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk.* Morgan Kaufmann, 2007.

[54] A. Jacobvitz, A. Hilton, and D. Sorin, "Multi-program benchmark definition," in *Proceedings International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015.

[55] Y. Jiang, K. Tian, and X. Shen, "Combining locality analysis with online proactive job co-scheduling in chip multiprocessors," in *HiPEAC*, 2010.

[56] A. Joy *et al.*, "Analog-DFE-based 16Gb/s SerDes in 40nm CMOS that operates across 34dB loss channels at Nyquist with a baud rate CDR and 1.2Vpp voltage-mode driver," in *ISSCC*, 2011.

[57] A. Kazmi, "Non-transparent bridging makes PCI-Express HA friendly," in *EE Times*, 2003.

[58] ——, "Minimizing PCI Express power consumption," in *PCI-SIG Developers Conference*, 2007.

[59] T. Li, D. Baumberger, D. Koufaty, and S. Hahn, "Efficient operating system scheduling for performance-asymmetric multi-core architectures," in *SC*, 2007.

[60] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. Reinhardt, and T. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," in *Proceedings of the 36th International Symposium on Computer Architecture (ISCA).* ACM/IEEE, 2009.

[61] K. Lim, Y. Turner, J. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. Wenisch, "System-level implications of disaggregated memory," in *Proceedings of the 18th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2012.

[62] M. Liu and T. Li, “Optimizing virtual machine consolidation performance on numa server architecture for cloud workloads,” in *Proceedings of the 41st Annual International Symposium on Computer Architecture (ISCA).* ACM/IEEE, 2014.

[63] Q. Llull, S. Fan, S. Zahedi, and B. Lee, “Cooper: Task colocation with cooperative games,” in *Proceedings of the 23rd IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[64] D. Lo, L. Chengn, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, “Heracles: improving resource efficiency at scale,” in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA).* ACM/IEEE, 2015.

[65] H. Löf and S. Holmgren, “affinity-on-next-touch: increasing the performance of an industrial pde solver on a cc-numa system,” in *Proceedings of the 19th annual international conference on Supercomputing (SC).* ACM, 2005.

[66] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein, “Graphlab: A new framework for parallel machine learning,” *arXiv preprint arXiv:1408.2041*, 2014.

[67] L. .Lu, H. Zhang, E. Smirni, G. Jiang, and K. Yoshihira, “Predictive vm consolidation on multiple resources: Beyond load balancing,” in *Proceedings International Symposium on Quality of Service (IWQoS).* IEEE, 2013.

[68] K. Malladi, F. Nothaft, K. Periyathambi, B. Lee, C. Kozyrakis, and M. Horowitz, “Towards energy-proportional datacenter memory with mobile DRAM,” in *Proceedings of the 39th International Symposium on Computer Architecture (ISCA).* ACM/IEEE, 2012.

[69] K. Malladi, I. Shaeffer, L. Gopalakrishnan, D. Lo, B. Lee, and M. Horowitz, “Rethinking DRAM power modes for energy proportionality,” in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012.

[70] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. Soffa, “Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations,” in *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011.

[71] J. McCalpin, “TACC Ranger node local and remote memory latency tables,” 2012. [Online]. Available: http://blogs.utexas.edu/jdm4372/2012/07/26/amd-opteron-remote-memory-latency-table/

[72] D. Meisner, B. Gold, and T. Wenisch, “PowerNap: Eliminating server idle power,” in *Proceedings of the 14th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[73] D. Meisner, J. Wu, and T. Wenisch, “BigHouse: A simulation infrastructure for data center systems,” in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2012.

[74] H. Michael, “recommenderlab: A framework for developing and testing recommendation algorithms,” 2011.

[75] Micron, “Calculating memory system power for DDR3,” in *Technicalj Note TN-41-01*, 2007.

[76] H. Moulin, *Fair division and collective welfare.* MIT Press Cambridge, 2004.

[77] O. Mutlu and T. Moscibroda, “Stall-time fair memory access scheduling for chip multiprocessors,” in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2007.

[78] ——, “Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems,” in *Proceedings of the 35th International Symposium on Computer Architecture (ISCA)*, 2008.

[79] K. Nesbit, N. Aggarwal, J. Laudon, and J. Smith, “Fair queueing memory systems,” in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2006.

[80] N. Nisan, T. Roughgarden, E. Tardos, and V. Vazirani, *Algorithmic game theory.* Cambridge University Press Cambridge, 2007, vol. 1.

[81] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, “Scale-out NUMA,” in *Proceedings of the 19th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[82] D. Parkes, A. Proceedingsccia, and N. Shah, “Beyond dominant resource fairness: Extensions, limitations and indivisibilities,” in *Proceedings Conference on Electronic Commerce*, 2012.

[83] A. Patel, F. Afram, S. Chen, and K. Ghose, “MARSSx86: A full system simulator for x86 CPUs,” in *Proceedings of the 48th Design Automation Conference (DAC)*. ACM, 2011.

[84] PLX Technology, “PLX PCIe switch power consumption,” in *PLX Technology White Paper*, 2008.

[85] Q. Pu, H. Li, M. Zaharia, A. Ghodsi, and I. Stoica, "Fairride: near-optimal, fair cache sharing," in *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[86] M. Qureshi and Y. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2006.

[87] C. Ranger, R. Raghuramana, A. Penmetsa, G. Bradsi, and C. Kozyrakis, "Evaluating MapReduce for multi-core and multiprocessor systems," in *Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2007.

[88] J. Rao, K. Wang, X. Zhou, and C. Xu, "Optimizing virtual machine scheduling in NUMA multicore systems," in *Proceedings of the 19th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[89] J. Regula, "Using non-transparent bridging in PCI Express systems," in *PLX Technology White Paper*, 2004.

[90] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt, "Google-wide profiling: A continuous profiling infrastructure for data centers," *IEEE Micro*, 2010.

[91] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, 2011.

[92] A. Roth, "Introduction to the Shapley value," in *The Shapley value: Essays in honor of Lloyd S. Shapley.* Cambridge University Press, 1988.

[93] W. Saad, Z. Han, M. Debbah, A. Hjorungnes, and T. Basar, "Coalitional game theory for communication networks," *Signal Processing Magazine, IEEE*, 2009.

[94] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Item-based collaborative filtering recommendation algorithms," in *Proceedings of the 10th international conference on World Wide Web.* ACM, 2001.

[95] L. Shea, "Performance impact: The cost of NUMA remote memory access," 2012. [Online]. Available: http://sqlblog.com/blogs/linchi_shea/archive/2012/01/30/performance-impact-the-cost-of-numa-remote-memory-access.aspx

[96] A. Snavely and D. Tullsen, "Symbiotic job scheduling for a simultaneous multithreading processor," in *Proceedings the 9th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[97] J. Stamper, A. Niculescu-Mizil, S. Ritter, G. Gordon, and K. Koedinger, "Algebra i 2006-2007. challenge data set from kdd cup 2010 educational data mining challenge," http://pslcdatashop.web.cmu.edu/KDDCup/downloads.jsp.

[98] L. Tan, J. Mars, X. Zhang, R. Hagmann, R. Hundt, and E. Tune, "Optimizing Google's warehouse scale computers: The NUMA experience," in *Proceedings of the 19th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[99] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. Soffa, "The impact of memory subsystem resource sharing on datacenter applications," in *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*. ACM/IEEE, 2011.

[100] A. Udipi, N. Muralimanohar, R. Balasubramonian, A. Davis, and N. Jouppi, "LOT-ECC: Localized and tiered reliability mechanisms for commodity memory systems," in *Proceedings of the 39th International Symposium on Computer Architecture (ISCA)*. ACM/IEEE, 2012.

[101] H. Varian and J. Repcheck, *Intermediate microeconomics: a modern approach*. WW Norton & Company New York, 2010, vol. 6.

[102] V. Vavilapalli, A. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing (SOCC)*, 2013.

[103] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: efficient performance prediction for large-scale advanced analytics," in *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[104] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum, "Operating system support for improving data locality on cc-numa compute servers," in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 1996.

[105] T. Vogelsang, "Understanding the energy consumption of dynamic random access memories," in *Proceedings of the 43th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010.

[106] D. Wang, D. Irani, and C. Pu, "Evolutionary study of web spam: Webb spam corpus 2011 versus webb spam corpus 2006," in *In Proceedings of 8th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*, 2011.

[107] Q. Wang and B. C. Lee, "Modeling communication costs in blade servers," in *Proceedings of the Workshop on Power-Aware Computing and Systems.* ACM, 2015, pp. 11–15.

[108] X. Wang and J. Martínez, "Xchange: A market-based approach to scalable dynamic multi-resource allocation in multicore architectures," in *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[109] X. Wang and J. F. Martínez, "Rebudget: Trading off efficiency vs. fairness in market-based multicore resource allocation via runtime budget reassignment," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM, 2016, pp. 19–32.

[110] R. Williams, "Server memory roadmap," in *JEDEC Server Memory Forum*, 2011.

[111] F. Wu and L. Zhang, "Proportional response dynamics leads to market equilibrium," in *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing.* ACM, 2007.

[112] D. Xu, C. Wu, and P.-C. Yew, "On mitigating memory bandwidth contention through bandwidth-aware scheduling," in *Proceedings International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[113] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[114] D. Yoon and M. Erez, "Virtualized and flexible ECC for main memory," in *Proceedings of the 15th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[115] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European conference on Computer systems (EuroSys)*, 2010.

[116] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I.Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[117] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[118] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing (HotCloud)*, 2010.

[119] S. M. Zahedi and B. Lee, "REF: Resource elasticity fairness with sharing incentives for multiprocessors," in *Proceedings of the 19th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[120] S. M. Zahedi and B. C. Lee, "Sharing incentives and fair division for multiprocessors," *IEEE Micro*, vol. 35, no. 3, pp. 92–100, 2015.

[121] L. Zhang, "Proportional response dynamics in the fisher market," *Theoretical Computer Science*, vol. 412, no. 24, pp. 2691–2698, 2011.

[122] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

Biography

Qiuyun Llull was born on September 4th, 1989 in Yichang, Hubei Province, China. In 2010, she received her Bachelor of Engineering in Opto-electronic Information Engineering at Huazhong University of Science and Technology, China. She was awarded as an outstanding graduate in 2010. In 2012, she received her Master of Science degree from Université Paris Sud (Paris XI), France, in Embedded Systems.

She began studying under Prof. Benjamin C. Lee at the Duke University System and Architecture Integration Laboratory (SAIL) in September 2012. Her research spans the fields of performance modeling and analysis for datacenter applications, microeconomic models for system management policy design. Her most recent research "Task Colocation with Cooperative Games" [63] was published in HPCA 2017. Her previous work, "Modeling Communication Costs in Blade Servers"[107], was selected as a Best Paper in HotPower 2015; she has also presented a tutorial series on Datacenter Simulation Methodologies in ISCA 2015, ISPASS 2015, and MICRO 2014. In 2017, she received her Doctor of Philosophy degree from Duke University in Computer Engineering.