2017 IEEE International Symposium on High Performance Computer Architecture

Cooper: Task Colocation with Cooperative Games

Qiuyun Llull, Songchun Fan, Seyed Majid Zahedi, Benjamin C. Lee
Duke University
{*qiuyun.wang, songchun.fan, seyedmajid.zahedi, benjamin.c.lee*}*@duke.edu*

***Abstract*—Task colocation improves datacenter utilization but introduces resource contention for shared hardware. In this setting, a particular challenge is balancing performance and fairness. We present** Cooper**, a game-theoretic framework for task colocation that provides fairness while preserving performance.** Cooper **predicts users' colocation preferences and finds stable matches between them. Its colocations satisfy preferences and encourage strategic users to participate in shared systems. Given** Cooper**'s colocations, users' performance penalties are strongly correlated to their contributions to contention, which is fair according to cooperative game theory. Moreover, its colocations perform within 5% of prior heuristics.**

***Keywords*-Datacenter Management, Task Colocation, Interference, Performance Prediction, Fairness, Game Theory**

I. Introduction

Modern datacenters, with their increasingly parallel computation and increasingly capable machines, colocate small tasks on big servers. A task partially uses a server's resources, but all resources become available when the server is powered. When a server's large power costs are amortized over little work, energy efficiency suffers [1]. Colocating multiple tasks on each server increases efficiency but introduces contention for shared resources such as last-level cache capacity and memory bandwidth [2], [3], [4].

Current colocation policies manage 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 [5], [6] or use architectural insights to pair applications with complementary resource demands [2], [4].

Performance-centric policies are insufficient for privately shared systems. 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. However, 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 [7]. Real-world users are selfish and rational [8], [9], an observation that has motivated numerous game-theoretic perspectives on systems management [10], [11], [12], [13], [14]. 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. Fairness addresses these challenges, ensuring system integrity and stability.

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 [15], [16]. We say colocations are fair when similar tasks suffer similar performance losses. When tasks are dissimilar, the relevant differentiator is contentiousness. Thus, users' performance losses from colocation should increase with their contributions to contention.

In addition to fairness, we seek colocations that satisfy user preferences and enhance system stability. Users prefer less contentious co-runners and smaller performance losses. Satisfied preferences enhance stability by reducing users' incentives to find better co-runners or break into separate, less efficient systems.

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

2378-203X/17 $31.00 © 2017 IEEE
DOI 10.1109/HPCA.2017.22

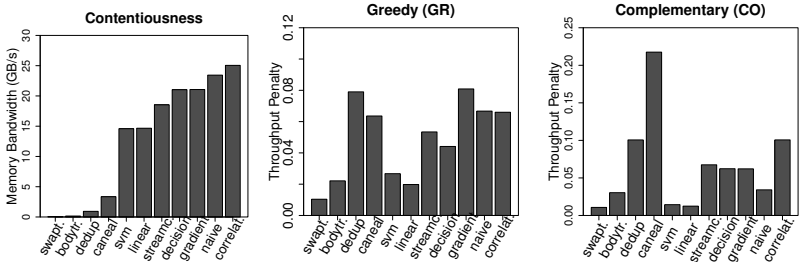

Figure 1: Unfair colocations show no link between contentiousness and penalties. We colocate 1000 jobs drawn randomly from pool of jobs. Pairs of jobs share last-level cache and memory bandwidth. Colocation penalties are averaged over those that include a particular job (e.g., `bodytrack`).

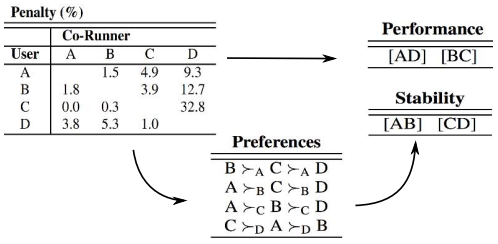

Figure 2: Pursuing performance minimizes system penalties. Pursuing stability satisfies users' preferences for lower penalties. Data for four users: (A) `x264`, (B) `fluidanimate`, (C) `decision-tree`, (D) `regression`.

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.

II. Case for Fair Colocation

We approach fair colocation from a game-theoretic perspective, describing strategic situations that arise when strategic users share systems. Cooperative game theory prescribes the fair division of costs that arise from interactions between strategic agents [17]. Solutions to these games reconcile agents' divergent preferences and produce stable outcomes [18], [19]. We use such theories to design and analyze colocation policies.

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;[1] 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 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.

Fairness and Contention. We argue that a colocation's performance penalties are attributed fairly when more contentious users incur larger penalties. In practice, such fair attribution encourages participation. Suppose Alice's job is contentious and Bob's is not. If Bob contributes 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 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 notion of fairness is justified by the Shapley value in cooperative game theory [20]. Shapley determines each agent's fair share of a common outcome based on her contributions. Equation 1 shows the Shapley calculation. When applied to colocation, ϕ_i is agent i's fair share of penalty p, which depends on the agents from N that form colocation S.

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

Agent i's marginal contribution to 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 ways that colocations could form.[2]

Shapley 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.

Preferences and Stability. We pursue fairness through stable colocations, which satisfy user preferences and

[1] See §III for the formal definition of preference.

[2] See Appendix A for Shapley example.

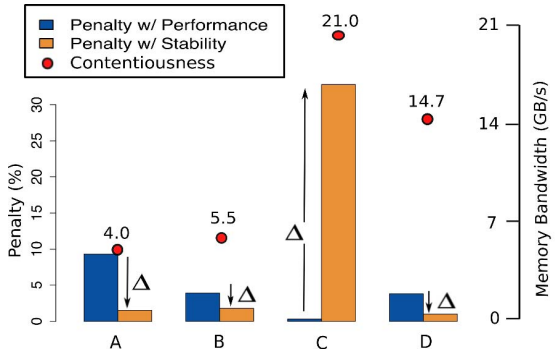

Figure 3: Stability enhances fairness. Bars show penalty (throughput loss) under performance- and stability-centric colocation. Dots show contentiousness (bandwidth demand). Data for four users: (A) `x264`, (B) `fluidanimate`, (C) `decision-tree`, (D) `regression`.

strengthen system integrity. Figure 2 illustrates instability from existing policies. Suppose four users share two processors and compete for 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, stable colocations {AB, CD} satisfy three of four users' preferences – A, B and D's. No pair wants to break away to form their own subsystem.

Figure 3 indicates that pursuing stability enhances fairness whereas pursuing 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 contrast, stable policies more closely align penalties with memory intensity. The penalty for the most contentious user rises while those for less contentious users fall. Stability furthers the fair attribution of costs in shared systems.

III. 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.

A. 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

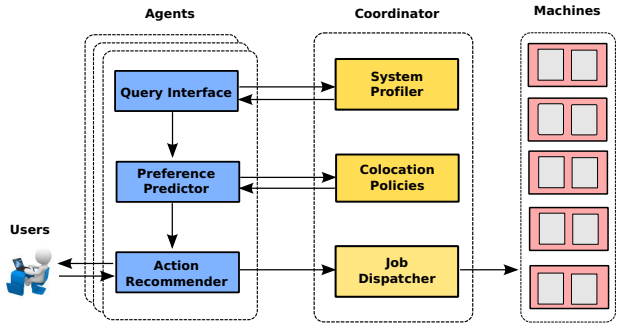

Figure 4: Agents act on users' behalf, playing the colocation game and interfacing with the system coordinator. The agents and the coordinator shield hardware complexity from human users.

times (i.e., minutes rather than seconds or milliseconds). If the system is heavily loaded, jobs queue for scheduling.

Figure 4 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, 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.

B. 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.

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
```

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.

C. 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 inspiration from stable algorithms for marriage [21] and roommate assignment [19], 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.

Preferences	Round	Propose	Accept	Reject
$m_1 : c_1 \succ c_2 \succ c_3$	1	$m_1 \to c_1$	$c_1 - m_3$	m_1
$m_2 : c_3 \succ c_1 \succ c_2$		$m_2 \to c_3$	$c_3 - m_2$	
$m_3 : c_1 \succ c_2 \succ c_3$		$m_3 \to c_1$	$c_2 -$	
$c_1 : m_2 \succ m_3 \succ m_1$	2	$m_1 \to c_2$	$c_2 - m_1$	
$c_2 : m_3 \succ m_1 \succ m_2$				
$c_3 : m_2 \succ m_1 \succ m_3$				

Figure 5: Stable marriage with compute- and memory-intensive jobs.

The procedure permits a parallel implementation. 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 [18]. 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 $w' \succ_m w$.

Adapting Partitions and Proposals. 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 [22]. 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.

Marriage Example. Figure 5 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 [19]. 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. We extend the roommates algorithm with heuristics when no stable solution exists. Perfect stability, defined by the absence of blocking pairs, may be impossible when an agent may pair with any other. When Irving's algorithm terminates with no solution, we greedily pair unmatched agents to minimize their individual 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.

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.

IV. Cooper Design

Figure 6 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 modules. The system profiler responds to queries with a database of performance measurements.

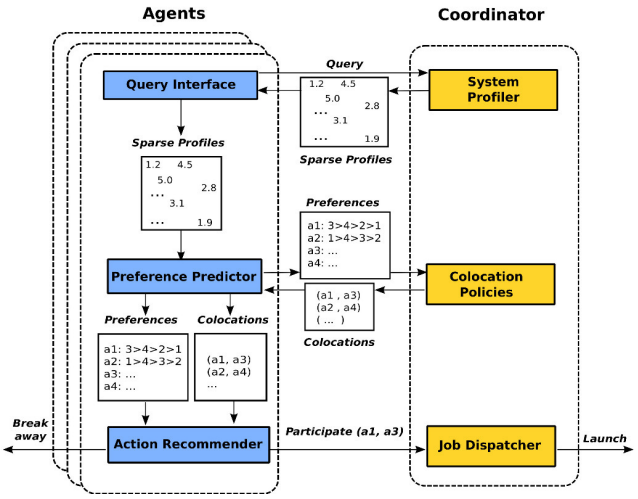

Figure 6: Cooper Colocation Framework

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.

A. 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.

Implementation. `Cooper` predicts preferences using an R library – `recommenderlab` [23]. 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%.

B. 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.

C. 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 [5]. 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.

D. 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 [24]. 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.

V. Experimental Methodology

Workloads. Table I summarizes evaluation benchmarks from Spark [29] and PARSEC 2.0 [30], which are representative of batch computation and data analytics. Methods

ID. Name	Application	Dataset	GBps
Apache Spark			
1. Correlation	Statistics	kdda'10 [25]	25.05
2. DecisionTree	Classifier	kdda'10	21.03
3. Fpgrowth	Mining	wdc'12 [26]	10.06
4. Gradient	Classifier	kdda'10	21.06
5. Kmeans	Clustering	uscensus [27]	0.32
6. Regression	Classifier	kdda'10	14.66
7. Movie	Recommender	movielens [28]	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 I: Application configurations, datasets, and memory intensity.

for multiprogrammed benchmarking vary [31]. 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 [6], [32].

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 I. 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 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.

VI. 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.

A. Fairness and Desiderata

Fair Attribution of Costs. Figure 7 evaluates 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 matches overrides preferences and induces unfairness.[3]

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 8 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 9 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 to contention, a fair outcome.

Stable Colocations. Figure 10 counts agents that recommend breaking away from assigned colocations for new,

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

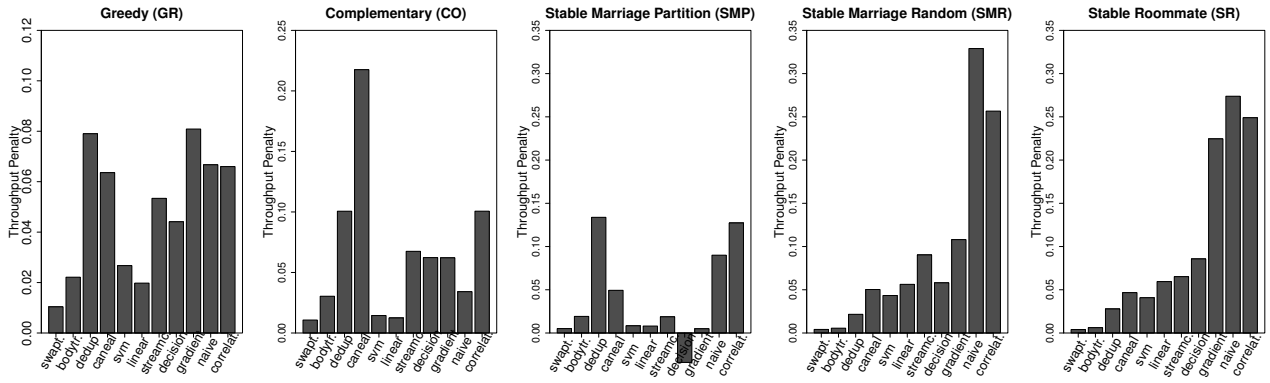

Figure 7: (a) and (b) show contention-induced performance losses from conventional colocation policies. (c)-(e) show losses from stable colocation policies. Jobs are ordered by increasing contentiousness on x-axis, as shown in Figure 1. Data is averaged over varied colocations when 1000 randomly sampled jobs share system.

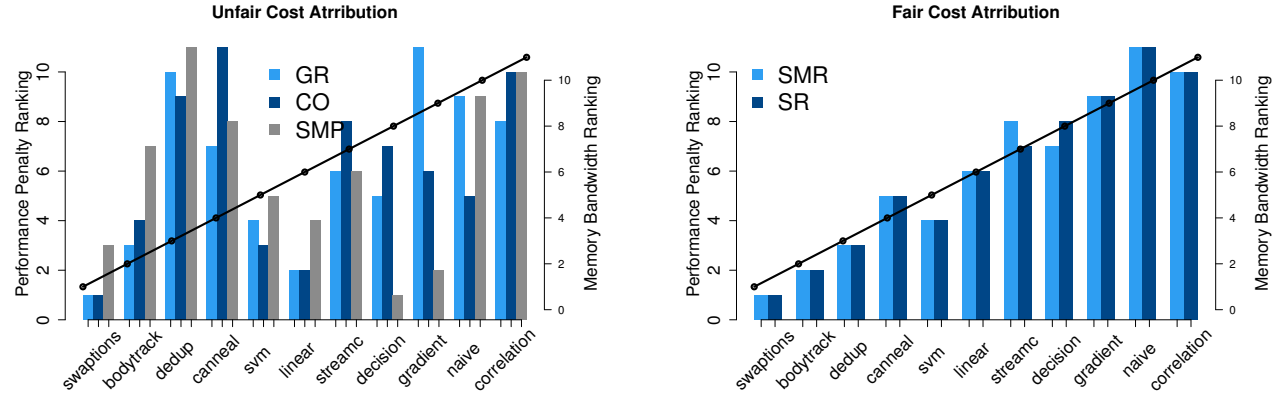

Figure 8: Correlation between ranked performance penalties (bars) and bandwidth demands (line). When the bars track the line, colocations are fair. See Figures 7 for absolute measures of performance and bandwidth.

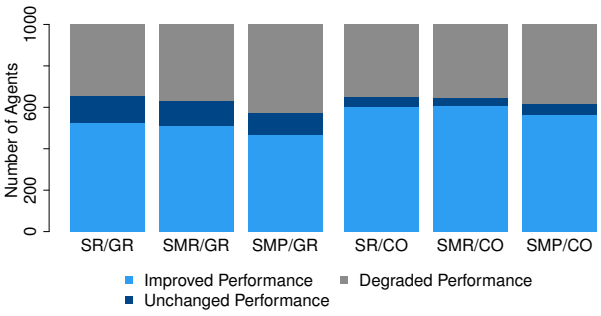

Figure 9: Performance impact when adopting cooperative game (S*) instead of performance-centric policies (GR, CO). Data is averaged over 10 populations, each with 1000 randomly sampled jobs.

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.

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 de-

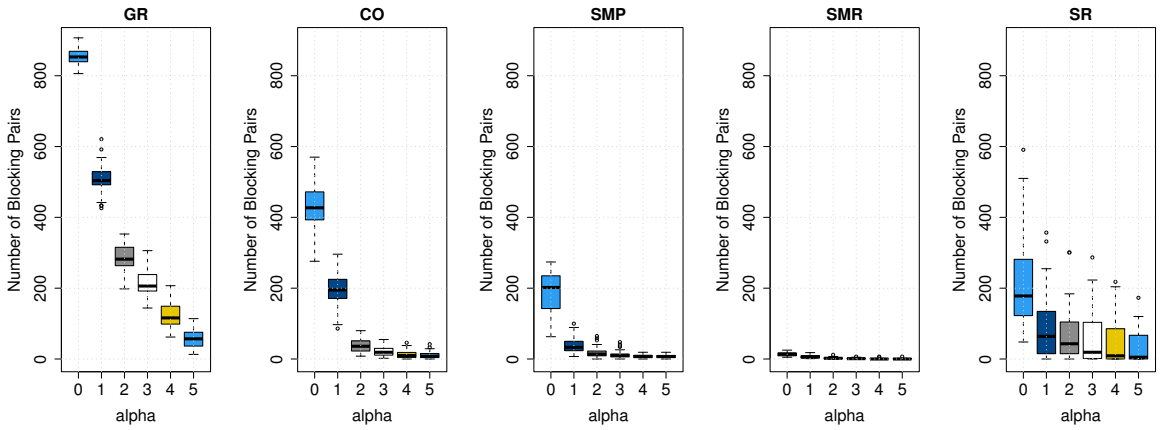

Figure 10: Stability analysis, which measures the number of blocking pairs (y-axis) for varied policies and α (x-axis), the minimum benefit for which an agent breaks away. When α=2%, agents break away for new colocations that improve both agents' performance by 2%. Here, we show data distributions and boxplots for 50 populations, each with 1000 randomly sampled jobs.

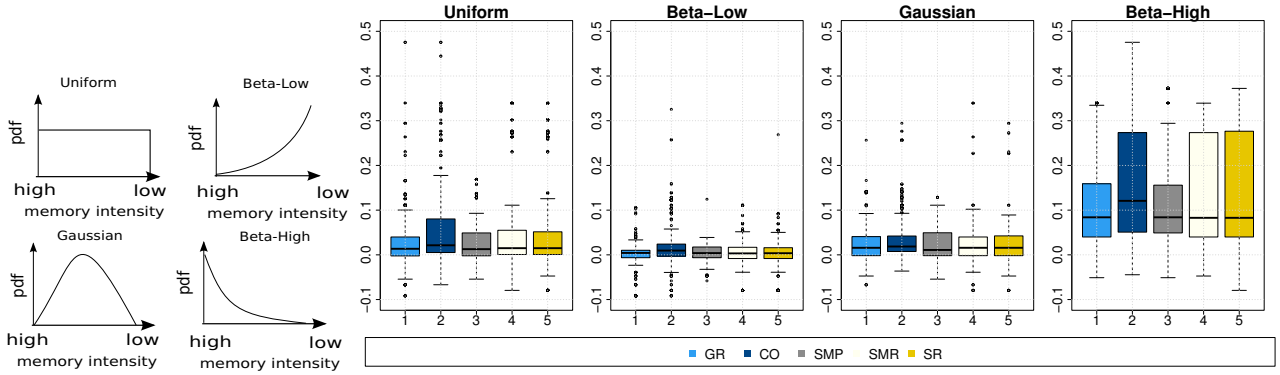

Figure 11: Performance penalties measured for varied colocation polices (GR, CO, SMP, SMR, SR) and workload mixes (Uniform, Beta-Low, Gaussian, Beta-High).

livers 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.

B. Performance and Sensitivity

Figure 11 presents 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 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 [22]. 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.

C. Preference Prediction

Figure 12 evaluates collaborative filtering and the accuracy of its predicted preferences. The rank coefficient

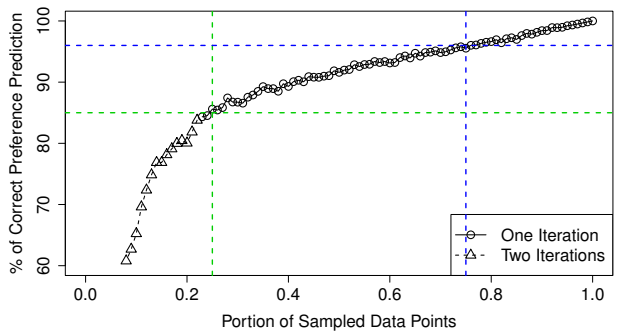

Figure 12: Prediction accuracy, which evaluates the percentage of correctly predicted preferences (Equation 2). x-axis shows various sample ratios.

τ compares a predicted list against the true list, counting inconsistencies.

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

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 12 indicates the accuracy of collaborative filtering improves with more data, starting at 83% with 25% of colocations profiled and rising to 95% with 75% profiled. With such accuracy, our stable policies deliver the same desiderata whether using oracular knowledge or collaborative filtering.

D. Scalability

Figure 13 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.

VII. 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 [33], [34], [35], [36]. We develop a system-level colocation 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

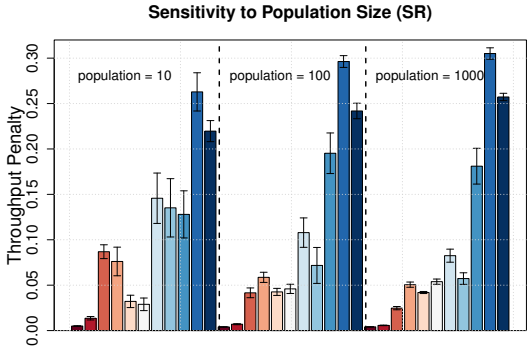

Figure 13: Scalability analysis and SMR fairness as the number of agents increases. Link between contentiousness and penalty is weak in small systems. In larger systems, more contentious jobs have larger penalties.

al. propose Dominant Resource Fairness to allocate cores and memory [11]. Zahedi et al., proposes Resource Elasticity Fairness using Cobb-Douglas utility function for cache and memory bandwidth allocation [13]. DRF and REF guarantee sharing incentives, Pareto efficiency, envy-freeness, and strategy-proofness. Grandl et al. propose Tetris [37], 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 [5], [38]. Delimitrou et al. models interference and machine heterogeneity with recommenders [6]. Multiple studies schedule complementary workloads on chip multiprocessors [3], [39], [40], [41], [42].

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 [43]. In networks, Feigenbaum et al. use cooperative games to attribute shared bandwidth costs during multicast transmission [44]. Han et al. use a repeated game that optimizes packet forwarding for strategic

and distributed users [45]. 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 [46]. In contrast, we bring cooperative games to datacenter colocation and seek solutions that balance fairness, stability, and performance for strategic users.

VIII. 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[47]. 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 [48]. In a similar spirit, `Cooper` operationalizes stable matching for future datacenters shared by strategic users.

Acknowledgments

We thank the anonymous reviewers 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

Appendix

We apply Shapley to motivate larger colocation penalties for more contentious users. Consider a simple model of colocation and its contention-induced penalties.

Users A, B, and C perform normally when alone but suffer penalties when colocated. Each user contributes interference

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 14: Shapley permutes users, calculates their contributions to penalties $\mathbf{M}$ and expected values ϕ over permutations.

$\{\mathrm{I_A} = 1,\ \mathrm{I_B} = 2,\ \mathrm{I_C} = 3\}$. 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 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 14 enumerates penalties and orderings for our example. Consider ordering {A, C, B}.

- A's marginal penalty is $\mathbf{M_A} = v(\mathrm{A}) - v(\emptyset) = 0$;
- 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 $\mathrm{I} = \{1,\ 2,\ 3\}$.

References

[1] 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*, pp. 1–154, 2013.

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

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

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

[5] 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 *International Symposium on Microarchitecture (MICRO)*, 2011.

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

[7] S. Zahedi and B. Lee, "Sharing incentives and fair division for multiprocessors," *IEEE Micro*, 2015.

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

[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] O. A. Ben-Yehuda, E. Posener, M. Ben-Yehuda, A. Schuster, and A. Mu'alem, "Ginseng: Market-driven memory allocation," in *International Conference on Virtual Execution Environments (VEE)*, 2014.

[11] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types." in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

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

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

[14] S. Fan, S. Zahedi, and B. Lee, "The computational sprinting game," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[15] Aristotle, *Nicomachean Ethics*.

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

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

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

[19] R. W. Irving, "An efficient algorithm for the stable roommates problem," *Journal of Algorithms*, pp. 577–595, 1985.

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

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

[22] K. Iwama and S. Miyazaki, "A survey of the stable marriage problem and its variants," in *Informatics Education and Research for Knowledge-Circulating Society*. IEEE, 2008.

[23] H. Michael, "recommenderlab: A framework for developing and testing recommendation algorithms," 2011.

[24] 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, Computer Society*, 2010.

[25] 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.

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

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

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

[29] 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 *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[30] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," Tech. Rep., 2008.

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

[32] D. Lo, L. Chengn, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: improving resource efficiency at scale," in *International Symposium on Computer Architecture (ISCA)*, 2015.

[33] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2007.

[34] 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 *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2010.

[35] K. Nesbit, N. Aggarwal, J. Laudon, and J. Smith, "Fair queuing memory systems," in *International Symposium on Microarchitecture (MICRO)*, 2006.

[36] M. Qureshi and Y. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *International Symposium on Microarchitecture (MICRO)*, 2006.

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

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

[39] Y. Jiang, K. Tian, and X. Shen, "Combining locality analysis with online proactive job co-scheduling in chip multiprocessors," in *International Conference on High Performance Embedded Architectures and Compilers*, 2010.

[40] M. Liu and T. Li, "Optimizing virtual machine consolidation performance on numa server architecture for cloud workloads," in *International Symposium on Computer Architecture (ISCA)*. IEEE, 2014.

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

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

[43] M. Dong, T. Lan, and L. Zhong, "Rethink energy accounting with cooperative game theory," in *International Conference on Mobile Computing and Networking (MobiCom)*, 2014.

[44] J. Feigenbaum, C. Papadimitriou, and S. Shenker, "Sharing the cost of muliticast transmissions (preliminary version)," in *Symposium on Theory of Computing (STOC)*, 2000.

[45] 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.

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

[47] E. Arkin, S. Bae, A. Efrat, K. Okamoto, J. Mitchell, and V. Polishchuk, "Geometric stable roommates," *Information Processing Letters*, 2009.

[48] 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.