Resource Allocation in Streaming Environments

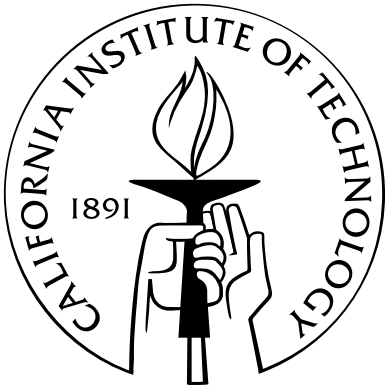
Thesis by

田璐 (Lu Tian)

In Partial Fulfillment of the Requirements

for the Degree of

Master of Science



California Institute of Technology

Pasadena, California

2006

(Submitted 26 May 2006)

Acknowledgements

First, I would like to thank my advisor, Professor K. Mani Chandy, for giving me tremendous support and guidance throughout the years of my graduate study. I feel extremely fortunate and honored to be his student and have the opportunity to work with him. He is not only an advisor and a friend, but also my role model. I especially want to thank him for the great effort he puts into helping his students find and achieve their personal goals in life.

I would also like to thank the members of my group—Mr. Agostino Capponi, Mr. Andrey Khorlin, and Dr. Daniel M. Zimmerman—for their help and collaboration, and their effort in creating an excellent working environment.

I must also acknowledge my good friends Patrick Hung, Kevin Tang, and Daniel M. Zimmerman for their great support, help, and friendship throughout the years, especially their excellent advice and inspiration regarding my thesis work.

Last but not least, I want to thank my family, my mother 白桂芝, my father 田德成, and my twin sister 田瑶 for always being there for me with remarkable support and understanding.

The research described in this thesis has been supported in part by the National Science Foundation under ITR grant CCR-0312778, and by the Social and Information Sciences Laboratory at the California Institute of Technology.

Abstract

The proliferation of the Internet and sensor networks has fueled the development of applications that process, analyze, and react to continuous data streams in a near-real-time manner. Examples of such stream applications include network traffic monitoring, intrusion detection, financial services, large-scale reconnaissance, and surveillance.

Unlike tasks in traditional scheduling problems, these stream processing applications are interacting repeating tasks, where iterations of computation are triggered by the arrival of new inputs. Furthermore, these repeated tasks are elastic in the quality of service, and the economic value of a computation depends on the time taken to execute it; for example, an arbitrage opportunity can disappear in seconds. Given limited resources, it is not possible to process all streams without delay. The more resource available to a computation, the less time it takes to process the input, and thus the more value it generates. Therefore, efficiently utilizing a network of limited distributed resources to optimize the net economic value of computations forms a new paradigm in the well-studied field of resource allocation.

We propose using a new performance model and resource reservation system as the solution space, and present two scheduling/resource allocation heuristics for processing streams in a distributed heterogenous computing environment to optimize economic value. Both heuristics are based on market mechanisms; one uses a centralized market and the other decentralized markets. We prove bounds on performance and present measurements to show that the performances of these two heuristics are near-optimal and significantly better than straightforward load-balancing heuristics.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The resource allocation problem is one of the oldest and most thoroughly studied problems in computer science. It is proven to be NP-complete [11, 13] and therefore computationally intractable. Thus, any practical scheduling algorithm presents a tradeoff between computational complexity and performance [6, 15]. There have been several good comparisons [3, 10, 23] of the more commonly used algorithms. The most common formulation of the problem in a distributed computing environment is: given a set of tasks, each associated with a priority number and a deadline, and a set of computing resources, assign tasks to resources and schedule their executions to optimize certain performance metrics. These optimizations include maximizing the number of tasks that can be processed without timing/deadline violations, minimizing the *makespan* (the total completion time of the entire task set), minimizing the aggregate weighted completion time, and minimizing computing resources needed to accommodate the executions of a set of tasks to meet their deadlines.

In the last decade, the proliferation of the Internet, the World Wide Web, and sensor networks has fueled the development of applications that process, analyze, and react to continuous data streams in a near-real-time manner. Each streaming application consists of a graph of multiple interacting and repeatedly executed processing units. These processing units perform particular types of processing on the incoming data streams—annotating or transforming the data in a stream or merging multiple streams—and publish the generated results as new streams or into persistent storage. For example, a financial company may have several such stream processing applications running concurrently. There could be an application continuously receiving and processing stock-tick prices, commodity prices, and foreign-exchange rates to detect arbitrage opportunities. There could be another application receiving stock-tick prices, total risk exposure and news story feeds, and using some model to monitor business performance and trigger alerts when the business enters dangerous critical situations. Furthermore, offline

post-processing on the data streams may be carried out for data mining, pattern discovery, and model building.

Stream processing applications differ from tasks in the traditional scheduling problem in the following aspects:

- Each is a repeating task triggered by inputs arriving at a specified rate, or arriving at random intervals with a known or unspecified probability distribution.
- Instead of having hard deadlines, tasks are elastic in quality of service (QoS), and the economic value of each computation depends on the time taken to execute it; for example, the value of delivering a stock tick stream to a user with a 10-second delay is less than that with a 1-second delay.
- There can be interdependences and communication among stream processing applications.

Therefore, the conventional models and their solutions are not applicable for the new problem:

- Associating each task with a single deadline is not suitable because the computations have elastic deadlines.
- Using a priority number to indicate a task's importance/value is not suitable because applications have elastic QoS and varying economic values depending on delay.
- The solution space of determining when and on what machine each task is to be executed is not suitable because the exact time at which each computation can start is not known *a priori*.
- The objective is not to minimize the makespan, as in the conventional multiprocessor scheduling problem, but rather to optimize the net economic value of the computations.

As in most real-time systems, it is critical to find the most efficient way to optimize the ongoing resource consumption of multiple, distributed, cooperating processing units. To accomplish this, the system must handle a variety of data streams, adapt to varying resource requirements, and scale with various input data rates. The system must also assist and coordinate the communication, the buffering/storage, and the input and output of the processing units.

In this thesis, we propose a new framework for solving resource allocation problems for streaming applications.

1.2 Thesis Structure

The remainder of this thesis is structured as follows:

In Chapter 2, we review the research literature on resource allocation problems in heterogeneous computing systems.

Chapters 3, 4, and 5 are devoted to the specific problem we are addressing: resource allocation for streaming applications.

In Chapter 3, we start by defining the computational fabric and streaming applications, then formulate our problem as an integer optimization problem and prove that it is NP-complete. We also propose using resource reservation systems as the solution space.

In Chapter 4, we present two market-based heuristics for building such resource reservation systems.

In Chapter 5, we evaluate the performances of our two market-based heuristics through computer simulations.

Finally, in Chapter 6, we present a summary of our results and a discussion of their applicability, and conclude with a discussion of future work.

Chapter 2

Resource Allocation on Heterogeneous Computing Systems

A computing system is said to be heterogeneous if differences in one or more of the following characteristics are sufficient to result in varying execution performance among machines: processor type, processor speed, mode of execution, memory size, number of processors (for parallel machines), inter-processor network (for parallel machines), *etc.*

The study of resource allocation and scheduling has transitioned from single processor systems to multiprocessor systems, from offline problems to online problems, from independent tasks to interacting tasks and from one-time tasks to repeated tasks. There is a rich body of research in this problem domain. Some resource allocation methods employ conventional computer science approaches, while others apply microeconomic theory and employ market or auction mechanisms. In this chapter, we review the studies conducted in this area.

2.1 Scheduling Heuristics

As described earlier, the most common formulation of the scheduling problem in a distributed computing environment is: given a set of tasks, each associated with a priority number and a deadline, and a set of computing resources, assign tasks to resources and schedule their executions to optimize certain performance metrics. There have been several good comparisons [3, 10, 23] of commonly used algorithms.

2.1.1 Engineering Heuristics

The heuristics presented in this section are for mapping/scheduling a set of tasks onto multiple processors. The objective of these heuristics is to minimize the makespan of the task set.

2.1.1.1 Bin-Packing Heuristics

Two bin-packing heuristics—*First Fit* and *Best Fit*—can be applied to solve the mapping problem. In this formulation, tasks are treated as balls and machines are treated as bins.

The First Fit heuristic is also called *Opportunistic Load Balancing* [4]. This is the simplest heuristic of all. It randomly chooses a task from the unmapped set of tasks and assigns it to the next available machine.

The Best Fit heuristic first calculates, for each task, the machine that can execute that task the fastest. It then assigns each task to its optimal machine.

2.1.1.2 Min-Min and Max-Min Heuristics

Min-Min Given a set of machines $\{M_1 \dots M_M\}$, the Min-Min heuristic starts with a set of unmapped tasks and repeats the following steps until the set of unmapped tasks is empty:

1. Let ET_{ij} be the execution time of task T_i on machine M_j. For each task T_i, calculate the minimum execution time ($MinET_i$) and its choice of machine $choice_i$ based on the current machine availability status:
   (a) $MinET_i = \min_{j=1}^{M} ET_{ij}$
   (b) $choice_i = \arg MinET_i$
2. Assign the task with the smallest $MinET_i$ to its choice of machine $choice_i$.
3. Remove this task from the unmapped set and update the availability status of its assigned machine.

The intuition behind this method is that the Min-Min way of assigning tasks to machines minimizes the changes to machine availability status. Therefore, the percentage of tasks assigned to their first choice machines is likely to be higher.

Max-Min This heuristic is similar to the Min-Min heuristic. It also starts with a set of unmapped tasks and calculates the minimum execution time and choice of machine for each task. However, instead of assigning the task with the smallest $MinET_i$ first, this heuristic assigns the task with the largest $MinET_i$ to its choice of machine first, then repeats the process of updating machine availability status and calculating $MinET_i$ for each task until the unmapped set is empty.

The intuition behind this method is that the Max-Min way of assigning tasks to machines minimizes the penalty incurred by tasks with longer execution times. It is also an attempt to

avoid the situation where all shorter tasks execute first, then longer tasks execute while some machines sit idle.

We can see that Min-Min and Max-Min both have advantages and disadvantages. This inspired the development of another heuristic, *Duplex*, to incorporate the advantages of both heuristics and avoid their pitfalls. The Duplex heuristic performs both Min-Min and Max-Min and chooses the better of the two results.

2.1.1.3 Genetic Algorithm and Simulated Annealing

Genetic Algorithm In genetic algorithms [14, 19], each chromosome is a vector that encodes a particular way of mapping the set of tasks to machines. Each position in this vector represents a task, and the corresponding entry represents the machine that task is mapped to, *e.g.*, if the i^{th} chromosome is $c_i = [0\ 1\ 1\ 2\ 1\ 3\ 0\ 4\ 2]$, $c_i[3] = 1$ means that task T_3 is assigned to machine M_1.

Each chromosome has a fitness value, which is the makespan that results from the mapping that chromosome encodes. The algorithm is as follows:

1. Generate the initial population, *e.g.*, a population of 200 chromosomes following a random distribution.
2. While no stopping condition is met:
   (a) **Selection**: Probabilistically duplicate some chromosomes and delete others. Chromosomes with good makespan have higher probability of being duplicated, whereas chromosomes with bad makespan have higher probability of being deleted.
   (b) **Crossover**: Randomly select a pair of chromosomes, then randomly select a region on those two chromosomes and interchange the machine assignments on the chosen region.
   (c) **Mutation**: Randomly select a chromosome, then randomly select a task on that chromosome and reassign it to a new machine.
   (d) **Evaluation**: Evaluate the following stopping conditions:
      i. The process has reached a certain number of total iterations.
      ii. No change in the best chromosome has occurred for a certain number of iterations.
      iii. All chromosomes have converged to the same mapping.
3. Output the best solution.

Simulated Annealing Simulated annealing [9, 18] also uses chromosomes to encode the mapping of tasks to machines. However, unlike genetic algorithms, which deal with a population of chromosomes, simulated annealing only considers one mapping at a time and uses a *mutation* process on that chromosome to improve its makespan. It introduces the notion of system temperature to probabilistically allow poorer solutions to be accepted at each iteration. This is an attempt to avoid being trapped at local optima and to better explore the solution space. The system temperature decreases with each iteration. The higher the temperature, the more likely it is that a worse mapping is accepted. Thus, worse mappings are likely to be accepted at system initiation, but less likely as the system evolves. The procedure is as follows:

1. Generate the initial mapping.
2. While no stopping condition is met:
   (a) **Mutation**: Randomly select a task on the chromosome and reassign it to a new machine.
   (b) **Acceptance decision**:
      i. If the new makespan is better, replace the new mapping with the old one.
      ii. Otherwise, probabilistically decide whether to accept the new mapping based on makespans and system temperature.
   (c) **Evaluation**: Evaluate the following stopping conditions:
      i. The system temperature approaches zero.
      ii. No change in the makespan has occurred for a certain number of iterations.
3. Output the best solution.

Genetic Simulated Annealing Genetic simulated annealing is a combination of the genetic algorithm and simulated annealing. It follows a procedure similar to that of the genetic algorithm with an annealing 'flavor' added: in the process of selection, mutation and crossover, the new chromosome is compared with the original chromosome and the decision of whether to accept the new mapping is made using the same probabilistic technique as that in simulated annealing.

2.1.2 Scheduling Periodic Tasks

A *periodic task* is a task that has an assigned period, is executed periodically, and must finish each execution before the start of the next period. That is, the deadline for the execution at each period is assigned to be the start of the next period. Periodic tasks are often scheduled preemptively using a class of algorithms known as *priority-driven* algorithms [10, 20, 23, 27]. The goal is to schedule a set of tasks such that all the deadlines are met. A task set is said to be schedulable if there exists a schedule such that all tasks can finish execution before their corresponding deadlines.

2.1.2.1 Scheduling Periodic Tasks on a Single Processor

Liu and Layland [23] proposed the *dynamic priority earliest-deadline-due (EDD)* algorithm. The priorities of tasks are based on their relative deadlines: the closer the deadline, the higher the priority. As its name implies, EDD schedules the task with the earliest deadline (highest priority) first. They proved that EDD is optimal among all scheduling algorithms (if a task set is schedulable, then EDD achieves a feasible schedule).

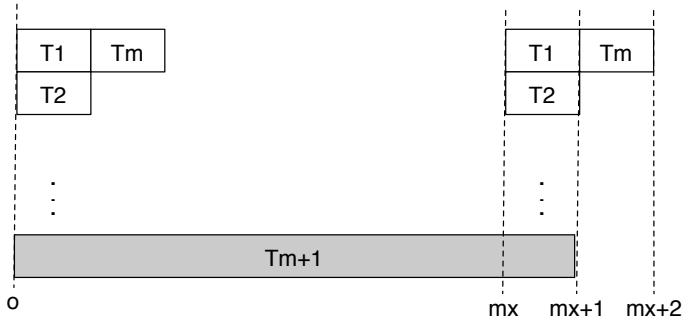
Another well-known priority-driven algorithm is the *rate monotonic (RM)* algorithm, also proposed by Liu and Layland. RM is a fixed-priority algorithm, where the priority for each task is defined by the fixed number $p_i = 1/T_i$ (the inverse of its period). Thus, the longer the period of the task, the lower priority it has. The task with the highest priority (shortest period) is scheduled first.

Liu and Layland presented sufficient schedulability conditions for RM: a set of N tasks is guaranteed to meet their deadlines on a single processor under RM scheduling if system utilization is no greater than $N(2^{\frac{1}{N}} - 1)$. This lower bound is called the *minimum achievable utilization*. It is a conservative but useful test for schedulability.

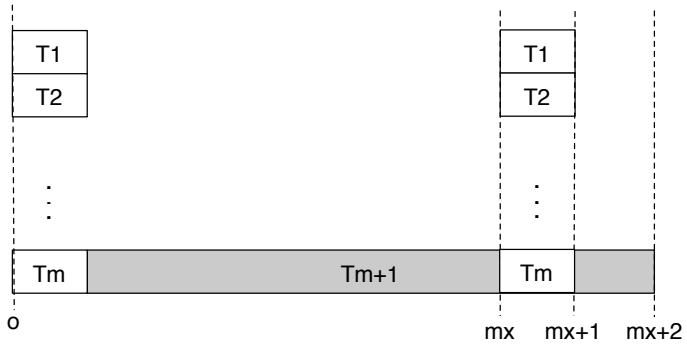
Due to its low computational requirement, the RM algorithm is one of the most popular choices for scheduling real-time periodic tasks on a single processor system.

2.1.2.2 Scheduling Periodic Tasks on Multiprocessor Systems

Although real-time tasks are expected to benefit greatly from the aggregated available resources multiprocessor systems can provide, scheduling these tasks on such systems is non-trivial. The objective here is to minimize the total number of machines needed to support the execution of a set of these tasks. There are two schemes for scheduling tasks on multiprocessor systems [7], *global scheduling* and *partitioning scheduling*.



(a) All tasks can be scheduled on m processors if T_{m+1} has one processor.



(b) T_{m+1} misses its deadline with RM using m processors.

Figure 2.1: An example showing poor RM scheduling performance

Global Scheduling In this scheme, all tasks are stored on a centralized server and the global scheduler selects the next task to be executed on a chosen processor. The task can be preempted and resumed on a different processor. One of the most commonly used algorithms for the global scheduling approach is the RM algorithm. However, Dhall and Liu [22] showed that this method can result in arbitrarily low utilization if the task set contains m tasks with compute time $C_i = 1$ and period $T_i = 2mx$ and one task with compute time $C_{m+1} = 2mx + 1$ and period $T_{m+1} = 2mx + 1$. The optimal solution is to schedule T_{m+1} by itself on a machine, T_1 and T_m on one machine, and $T_2 \ldots T_{m-1}$ each on a separate machine, as shown in Figure 2.1(a). However, the RM algorithm schedules the m easy tasks first because they have higher priority, and then the one hard task last (easy tasks are those with shorter periods, hard tasks are those with longer periods). As shown in Figure 2.1(b), RM requires at least $m+1$ processors for the task set. Total utilization of this set of tasks is

$$\begin{aligned} U &= \sum_{i=1}^{m} \frac{C_i}{T_i} + \frac{C_{m+1}}{T_{m+1}} \\ &= m\frac{1}{2mx} + \frac{2mx+1}{2mx+1} \\ &= \frac{1}{2x} + 1. \end{aligned}$$

As $x \to \infty$, the utilization becomes arbitrarily low. Furthermore, admission control is needed to guarantee that the tasks form a schedulable set.

Partitioning Scheduling In this scheme, the problem becomes twofold:

1. Partition the set of tasks and assign the subsets to machines.
2. Schedule task executions on each machine (*e.g.*, using the RM algorithm).

The first step, finding the optimal task assignment to machines, is equivalent to the bin-packing problem, which is proven to be NP-hard [21]. People have applied bin-packing heuristics such as First Fit and Next Fit to this problem, where the tasks are treated as balls and the machines are treated as bins. The decision of whether a bin is full is made based on the schedulability condition discussed previously. *Rate Monotonic Next Fit (RMNF)* and *Rate Monotonic First Fit (RMFF)* heuristics use Next Fit and First Fit heuristics for assigning the tasks to processors, and the RM heuristic for scheduling task executions on each machine. Both of them order tasks based on the lengths of their periods. RMNF assigns new tasks to the current processor until it is full (schedulability is violated), then moves on to a new processor. RMFF tries to assign a new task to a processor that is marked as full before assigning it to the current processor. A variation of RMFF is the *First Fit Decreasing Utility (FFDU)* method. In this method, instead of being sorted according to period, tasks are sorted according to *utility*, which is defined as the ratio between execution time and period. This ratio is also called the *load factor* of a task.

As mentioned earlier, the goal of these algorithms is to minimize the number of processors needed to accommodate the executions of all the tasks. To evaluate the performances of these methods, a worst case upper bound (ratio between the number of machines needed in the worst case and the optimal solution) is used. In summary, RMNF achieves a worst case upper bound of 2.67, RMFF achieves a worst case upper bound of 2.33, and FFDU achieves a worst case upper bound of 2.0.

2.2 State-of-the-Art Resource Management Systems

2.2.1 Spawn

Spawn [29] is an auction-based computational system that achieves two goals: (1) harness idle computational cycles from a distributed network of heterogeneous workstations, and (2) allocate idle resources to applications. It uses linear monetary funding as an abstraction for priority: the more funding, the higher priority. It employs a second-price auction market for allocating resources as follows:

1. Each task bids for the next available slot on the machine where it can finish earliest.

2. Each machine chooses from the available bids to maximize its revenue, awarding exclusive usage during the next timeslot to the highest bidding task while charging the second highest price.

3. The auction is not committed until the last possible moment.

This system emphasizes economic fairness over economic efficiency. A *fair* resource distribution is one in which each application is able to obtain a share of system resources proportional to its share of the total system funding. An *efficient* resource distribution maximizes aggregate utility (social welfare). Spawn uses a second-price auction. Thus, at any given time, there is only one auction open on each machine: bidding for the next available timeslot. This is a simple scheme, but it is not combinatorial, *i.e.*, each winning task, whose execution may span multiple timeslots, is guaranteed the next available timeslot but is not guaranteed future timeslots. Therefore, a situation can occur where a task needs K timeslots to execute, but is only able to get $k < K$ of them before exhausting its funds. This is a major disadvantage of the second-price single-item auction compared to combinatorial auctions. Furthermore, a winning task has exclusive use of the machine during its timeslot; situations can occur where less demanding tasks do not use their entire reservations, causing low utilization.

2.2.2 MSHN

MSHN (Management System for Heterogeneous Networks) [4] is a collaborative project funded by DARPA. Its main goal is to determine the best way to support executions of many different applications in a distributed heterogeneous environment. The system uses a conventional computer science approach: it employs a centralized scheduler and uses heuristics to solve the online scheduling problem.

MSHN uses a performance metric that takes deadline, priority, and versioning into account. Performance is measured as the weighted sum of completion times, where weights are the priority numbers assigned to the tasks. Formally, performance is $\sum(D_{ik}P_iR_{ik})$, where D_{ik} is 1 if task i is mapped to machine k and 0 otherwise, $P_i \in \{1,4,8,16\}$ is a priority assigned by the centralized scheduler, and R_{ik} is the time required to execute task i on machine k (entry (i,k) in the *expected time to completion* matrix).

This is a better measure of performance than the conventional optimization objective of minimizing completion time for the last (or an average) task. However, the three factors (deadline, priority and versioning) do not fully characterize how the cost of delaying the execution of a task varies with delay. In addition, the MSHN solution is centralized and may not scale

well enough for large systems. Finally, similar to other existing approaches, MSHN assumes that the exact time required to execute each task on each machine is known. In a streaming environment, where the arrival time of each new input is usually not known precisely *a priori*, this approach may not work well.

2.2.3 Tycoon

Tycoon [12] is a *proportional share abstraction*-based resource allocation system using an economic market-based approach, developed at HP Labs. Task owners choose computers to run their tasks and bid for resources on the chosen machines. Each bid is of the form $\{B_i, T_i\}$, where B_i is the total amount the task owner is willing to pay and T_i is the time interval she is bidding for. Machines make scheduling decisions by calculating a threshold price, B_k/T_k, for each timeframe (*e.g.*, 10 s) based on the collected bids. Any task with a bid above the threshold price gets a share of the machine resources proportional to the relative amount of its bid, $r_i = (B_i/T_i)/(B_k/T_k)$, and pays the machine based on its actual usage of the allocated resources, $payment_i = \min(q_i/r_q, 1)(B_i/T_i)$.

One drawback of this scheme is that, since payment is based on actual usage, a task's best strategy is to bid high on all machines to guarantee a share and ensure quick execution. This results in the following two problems:

1. Tasks bid high, get large amounts of resource allocated without fully utilizing it, and pay a small amount based on actual usage; this prevents other tasks from getting reasonable shares on the machine.

2. Tasks bid high and win on all machines, and continually switch to the machine requiring the lowest payment; this results in oscillation in the system.

Furthermore, the system works in an online environment, so the amount of resource allocated to each task changes dynamically depending on the other tasks' utilization of the machine resources and their bid amounts. Thus, there is no guarantee with respect to the resources available for the tasks running on each machine. To guarantee a given level of performance, a task must constantly monitor resource availability and change its bid accordingly.

2.2.4 Summary

This concludes our review of existing literature on the resource allocation problem. To the best of our knowledge, people have only studied this problem for traditional tasks, one-time or repeated, each characterized by a priority number and a deadline. No one has examined the resource allocation problem where tasks have elastic deadlines and the value of each task changes with the delay incurred in finishing the execution. In the following chapters, we focus on the resource allocation problem for this new class of tasks.

Chapter 3

Problem Formulation

There are several research projects studying various aspects of stream processing, *e.g.*, STREAM [5], Borealis [1], SMILE [16], and GATES [8]. However, their approaches for resource management are fundamentally different from the one presented here. For example, Borealis focuses on load-balancing: minimizing end-to-end latency by minimizing load variance.

In this chapter, we first describe the components of the problem domain: the computational fabric and streaming applications. Then, we formally describe our problem as an optimization problem and propose using resource reservation systems as the solution space. Finally, we provide a formal proof of the NP-completeness of the problem.

3.1 Computational Fabric

The computational fabric for stream processing is represented by a graph, where the nodes represent processors and the edges represent communication channels. Associated with each node and each edge is a set of parameters. The parameters of a node include the amount of memory, floating-point and fixed-point speeds, *etc.* The parameters of an edge include bandwidth and latency.

3.2 Streaming Application

Each streaming application can be represented as having three components:

1. One or more input flows.
2. A graph of interacting processing units.
3. One output flow.

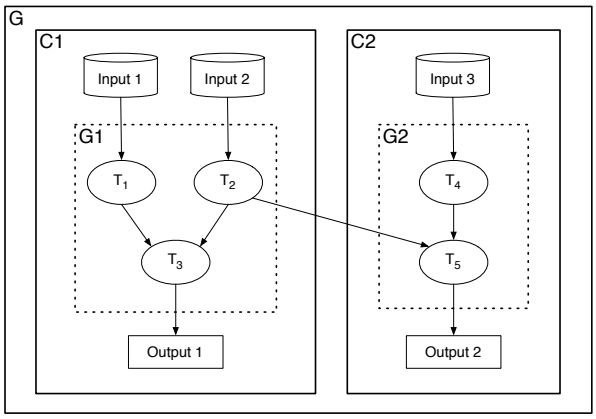


Figure 3.1: Two streaming applications

The interacting tasks are represented by a computation directed acyclic graph (DAG). The vertices of the graph represent tasks, and a directed edge represents flow of information from one task to another. We allow communication among applications, which means the output from one computation could be an input to another computation. Figure 3.1 shows a pair of interacting streaming applications. We model streaming applications using the following parameters:

1. A *persistence interval* $[T_{start_i}, T_{end_i}]$, where T_{start_i} is the time instant when the application starts receiving inputs and become active and T_{end_i} is the time instant when it stops receiving inputs and becomes inactive.

2. A *utility function* $U(r)$, which maps an amount of resource r to the value realized by processing the inputs with that guaranteed amount of resource.

For theoretical foundations about utility functions, refer to Mas-Colell, Whinston and Green [24]. In most cases, $U(r)$ is a concave nondecreasing function with parameters that depend on the application. Figure 3.2 shows the utility functions of three streaming applications.

3.3 Proposed Solution Space: Resource Reservation Systems

The more resource available to a streaming application, the quicker it can process its inputs, and thus the more utility it generates. Our objective is to allocate limited distributed resources to a set of streaming applications such that the total utility realized by all the applications is maximized.

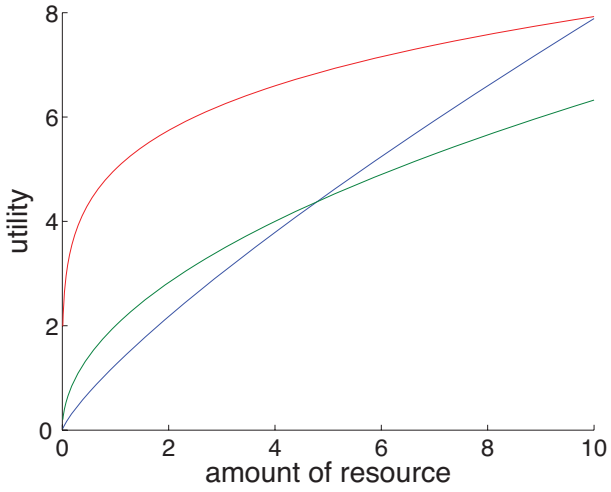


Figure 3.2: Utility functions

If we assume that each processing unit can be split and hosted on multiple machines, then the problem becomes the following continuous convex optimization problem, which is easily solved using standard optimization techniques:

$$\begin{aligned} \max \quad & \textstyle\sum_{i=1}^{N} U_i(r_i) \\ \text{subject to} \quad & r_i \geq 0, \\ & \textstyle\sum_{i=1}^{N} r_i \leq \sum C_j. \end{aligned}$$

However, computations on these streams often have substantial state; for example, a computation in a trading application maintains the state of the trade. Some operations on streams cannot be moved from one computer to another without also moving the states associated with the operations, so pinning operations to computers in a grid is an important step in the scheduling process. This turns the optimization problem into the following non-convex integer programming problem, which is NP-complete:

$$\begin{aligned} \max \quad & \textstyle\sum_{i=1}^{N} U_i(r_i) & & \\ \text{subject to} \quad & \textstyle\sum_{i=1}^{N} r_i x_{ij} \leq C_j & \text{for all} \;\; & j \in M \\ & r_i \geq 0 & \text{for all} \;\; & i \in N \\ & x_{ij} \in \{0,1\} & \text{for all} \;\; & i \in N, \; j \in M. \end{aligned}$$

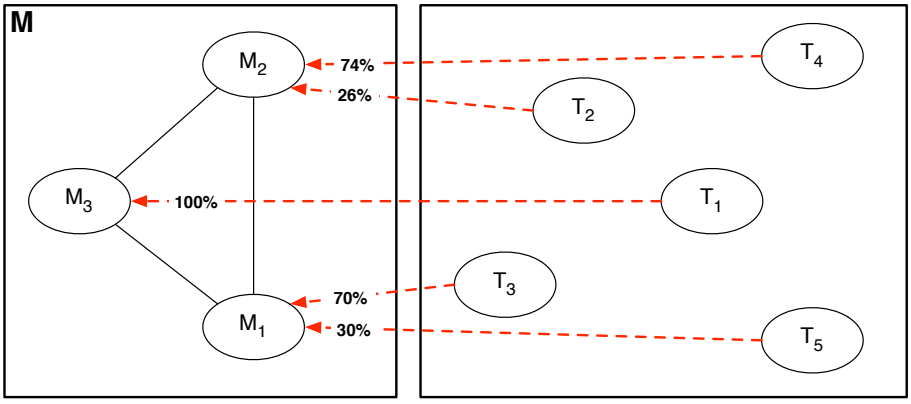


Figure 3.3: System view of the solution space

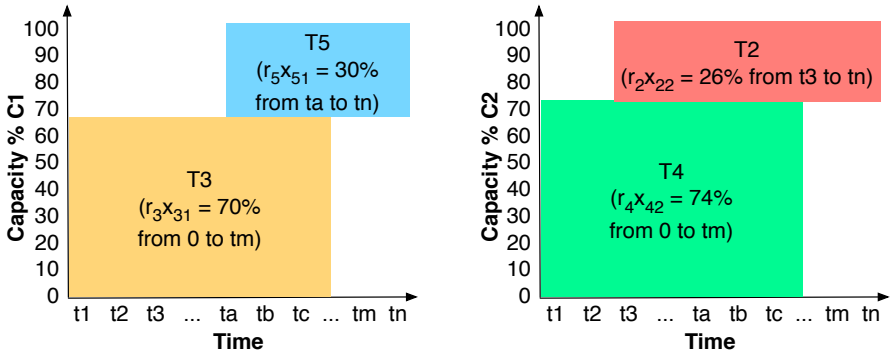


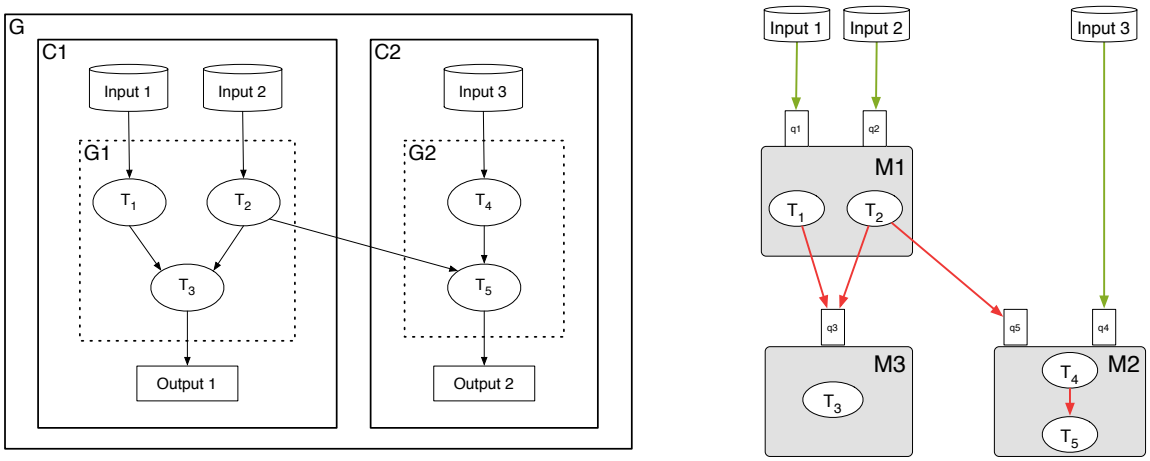
Figure 3.4: Per-machine view of the solution space

We propose using a resource reservation system to solve the problem in two steps:

1. Assign each processing unit T_i to one machine M_j.
2. Reserve a certain amount of M_j's resource r_{ij} for T_i's execution during its existence interval.

Figures 3.3 and 3.4 show a system view and a per-machine view of this solution space. Each resource reservation is of the form $(r_i, x_{ijt}, [T_{start_i}, T_{end_i}])$, where

- r_i denotes the amount of resource that is reserved for streaming application i;
- $x_{ijt} \in \{0, 1\}$ indicates whether stream i is assigned to machine j during time t;
- $[T_{start_i}, T_{end_i}]$ is the time interval during which streaming application i persists (requiring computing resources) in the system.



(a) problem example

(b) possible solution

Figure 3.5: Example of a mapping problem and a possible solution

Now we take a step back to see how to solve the overall mapping and scheduling problem:

1. Map streaming applications to machines.
   (a) Assign each streaming application to a machine.
   (b) Reserve a certain amount of that machine's resource for that streaming application.
2. Locally schedule the executions of the streaming applications assigned to each machine.

Figure 3.5(a) is our original pair of streaming applications, and Figure 3.5(b) shows a possible mapping for this problem. Tasks $T1$ and $T2$ are mapped to machine M1, and they each get a certain percentage of $M1$'s resource reserved for their computations; Task $T3$ is mapped to machine $M3$, and gets 100% of $M3$'s resource; Tasks $T4$ and $T5$ are mapped to machine $M2$, and each gets a certain percentage of $M2$'s resource. Each machine has one queue for each task located on it, to receive that task's inputs. At runtime, the machines make scheduling decisions about these queues based on several parameters, one of which could be the reservation amount for the corresponding task. The runtime scheduling step is described by Khorlin [17] in great detail.

Our focus is on the first step: mapping tasks to machines and making one reservation for each streaming application. We investigate a simpler version of the problem, where each streaming application consists of a single processing unit instead of a graph of interacting processing units and all streaming applications have the same existence interval. We propose the following problem formulation:

Let there be N streaming applications, each with a concave utility function $U_i(r_i)$, $i \in [1, N]$, and M machines, each with resource capacity C_j, $j \in [1, M]$. In our system, we assume that machine capacities are large relative to an individual streaming application's resource requirement.

The objective is to determine the allocation of resources, $\mathbf{r_i}$ (the amount of resource assigned to stream i during i's existence interval) and $\mathbf{x_{ij}}$ (a zero or one value indicating whether machine j's resource is allocated to stream i), that maximizes the total utility:

$$\begin{aligned} \max \quad & \textstyle\sum_{i=1}^{N} U_i(r_i) & \\ \text{subject to} \quad & \textstyle\sum_{i=1}^{N} r_i x_{ij} \le C_j & \text{for all } \ j \in M \\ & r_i \ge 0 & \text{for all } \ i \in N \\ & x_{ij} \in \{0,1\} & \text{for all } \ i \in N,\ j \in M. \end{aligned}$$

3.4 Hardness Proof

We can prove the NP-completeness of our problem by reduction from the Multiple Knapsack Problem (MKP). The MKP involves a set of M bins with capacities $C_1 \dots C_M$ and a set of N items with weights $w_1 \dots w_N$ and values $v_1 \dots v_N$. The objective is to find a feasible set of bin assignments $x_{ij} = \{0,1\}$ ($x_{ij} = 1$ if item i is assigned to bin j) that maximizes the sum of values of the items assigned to the bins. That is,

$$\begin{aligned} \max \quad & \textstyle\sum_{i=1}^{N} \sum_{j=1}^{M} v_i x_{ij} & \\ \text{subject to} \quad & \textstyle\sum_{i=1}^{N} w_i x_{ij} \le C_j & \text{for all } \ j \in M \\ & x_{ij} \in \{0,1\} & \text{for all } \ i \in N,\ j \in M \\ & \textstyle\sum_{j=1}^{M} x_{ij} \le 1 & \text{for all } \ i \in N. \end{aligned}$$

Theorem 1 *Our formulation of the resource allocation problem is NP-complete.*

Proof Given any instance of the MKP $\{C_j,\ j \in [1,M]$ and $(w_i, v_i),\ i \in [1,N]\}$, we can construct an instance of our problem as follows: Let there be M machines with capacities $C_1 \dots C_M$ and N tasks, all with the same existence interval $[t_0, t_k]$ and each with a constant utility function if the resource assigned to it has value at least w_i, *i.e.*, $U_i(\sum_{j=1}^{M} r_j x_{ij}) = v_i$ for $r_i \ge w_i$. Thus, the M machines correspond to the M bins, and the N tasks correspond to the N items. For each task, the resource threshold value $r_i \ge w_i$ corresponds to the weight of the item, and the constant utility $U_i(\sum_{j=1}^{M} r_j x_{ij}) = v_i$ corresponds to the value of the item. By assuming that all streaming applications have the same existence interval, we eliminate the time dimension, so x_{ijt} is the same across t and corresponds to the x_{ij} in the MKP indicating whether or not item/task i is

assigned to bin/machine j. Under this setup, an optimal solution to our problem must assign at most $r_i = w_i$ resource to streaming application i. Thus, the optimization problem can be formulated as follows:

$$\begin{aligned} \max \quad & \sum_{i=1}^{N} \sum_{j=1}^{M} v_i x_{ij} & \\ \text{subject to} \quad & \sum_{i=1}^{N} w_i x_{ij} \leq C_j & \text{for all } \ j \in M \\ & x_{ij} \in \{0,1\} & \text{for all } \ i \in N, \ j \in M \\ & \sum_{j=1}^{M} x_{ij} \leq 1 & \text{for all } \ i \in N. \end{aligned}$$

We can therefore convert any instance of the MKP to an instance of our problem. If we can solve any instance of our problem in polynomial time, then we can solve any instance of the MKP in polynomial time. Since the MKP is known to be NP-complete, our problem is also NP-complete.

□

Chapter 4

Market-Based Heuristics

In this chapter, we present two heuristics for building the resource reservation systems using the competitive market concept from microeconomics. Both heuristics need to determine the best allocation of a machine's resource to streams, which is equivalent to solving the following optimization problem:

$$\begin{aligned} \max \quad & \textstyle\sum_{i=1}^{N} U_i(r_i) \\ \text{subject to} \quad & \textstyle\sum_{i=1}^{N} r_i x_i \le \sum_{j=1}^{M} C_j \\ & r_i \ge 0 \qquad \text{for all } \ i \in N \\ & x_i \in \{0,1\} \qquad \text{for all } \ i \in N,\ j \in M,\ t \in T. \end{aligned}$$

This is equivalent to:

$$\begin{aligned} \min \quad & -\textstyle\sum_{i=1}^{N} U_i(r_i) \\ \text{subject to} \quad & \textstyle\sum_{i=1}^{N} r_i x_i \le \sum_{j=1}^{M} C_j \\ & r_i \ge 0 \qquad \text{for all } \ i \in N; \\ & x_i \in \{0,1\} \qquad \text{for all } \ i \in N,\ j \in M,\ t \in T. \end{aligned}$$

Since the objective function $\sum_{i=1}^{N} U_i(r_i)$ is concave and the constraints are linear, the problem is a convex optimization problem and can be solved using convex optimization techniques.

To avoid requiring each consumer to reveal its utility function to a centralized solver, we design a distributed method to solve the problem by using the market mechanism to solve its dual—the pricing problem. The pricing problem can be formulated as follows: There is a single market for resource with one supplier (the machine) and N consumers (the streaming applications). The supplier has an inelastic supply function $S = \mathit{capacity}$. Each consumer has a continuous concave utility function $U_i(r) = b_i \frac{r^{1-a_i}}{1-a_i}$.

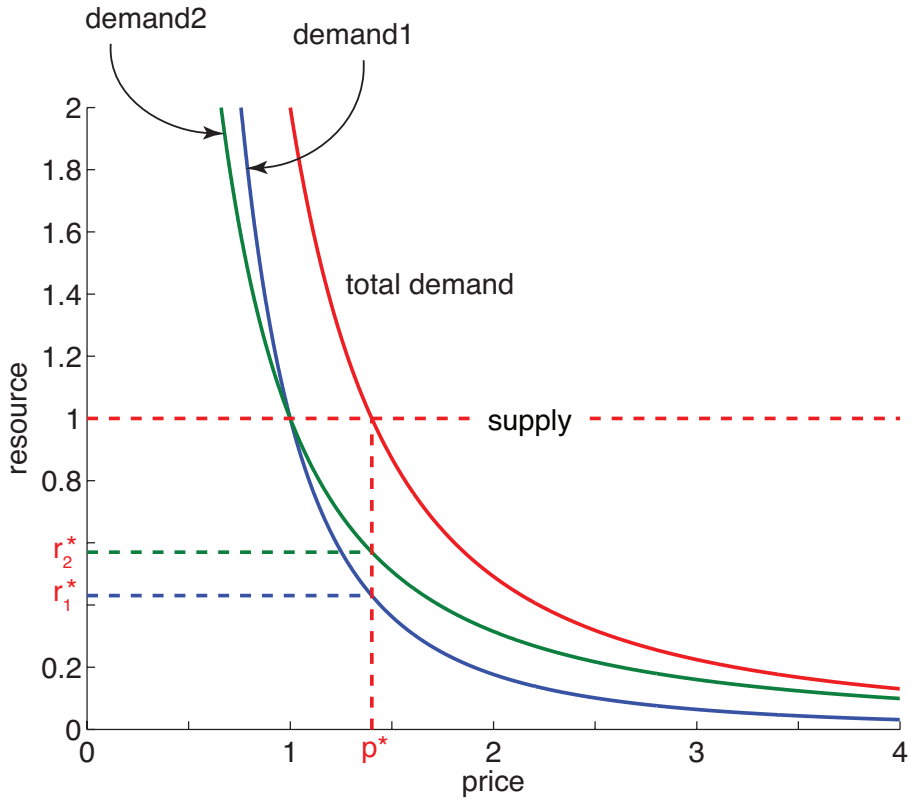


Figure 4.1: Demand, supply and optimal allocations

The supplier and the consumers interact with each other and participate in the price-adjustment process of the market:

1. The market sets an initial price.
2. While supply is not equal to total demand:
   (a) Each consumer reacts to price and adjusts its optimal consumption level by equating its marginal utility and the current market price; that is, it sets $U_i'(r) = r^{-a_i} = price$.
   (b) The supplier reacts to demand and updates the market price according to the excess demand.

The equilibrium price p^* is the price at which the total demand is equal to supply. At this equilibrium price each consumer has a corresponding demand r_i^*, which forms the solution to the resource allocation optimization problem.

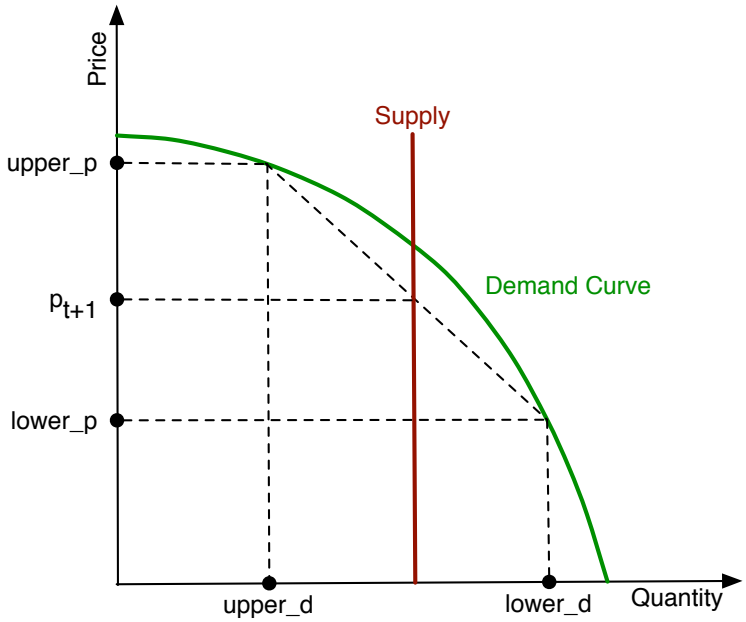
Figure 4.1 shows the demand-supply curves when there are two consumers in the market. The intersection of the total demand curve and the supply corresponds to the market equilibrium price. The demands of the consumers at that price, r_1^* and r_2^*, are the optimal solution to the resource allocation problem.

According to general equilibrium theory in microeconomics, there exists a unique equilibrium price for this convex optimization problem at which the market clears (total demand equals total supply) and achieves maximum social welfare (max $\sum_{i=1}^{N} U_i(r_i)$). The convergence property of the algorithm is also guaranteed [24].

The price update process (Figure 4.2) uses an interpolation method instead of the conventional step-size-based price adjustment. The interpolation method updates prices in two phases. In the first phase, it finds lower and upper bounds on the optimal price; the lower bound is a price when excess demand is greater than zero, and the upper bound is a price when excess demand is less than zero. When the bounds are found, it enters the second phase, where the new price is calculated by interpolating from the current price bounds. Experimental results show that this method results in a substantial performance improvement and cuts the number of iterations in the price adjustment process from thousands to fewer than one hundred.

```
update_price(excessDemand)
{
   /* find lower and upper prices to bound the optimal price */
   if (!(upperBoundFound && lowerBoundFound))
   {
      if (excessDemand > 0)
         p_{t+1} = 2p_t;
      if (excessDemand < 0)
         p_{t+1} = p_t/2;
   }
   else
      /* update the price using interpolation */
      p_{t+1} = lowerPrice +
         (lowerDemand − supply)(upperPrice − lowerPrice) / (lowerDemand − upperDemand);
}
```

(a) the price update algorithm



(b) pictorial description of the price update algorithm

Figure 4.2: The interpolation method for updating the market price

4.1 Single Market Resource Reservation System

The first resource reservation system we present is a three step process:

1. Use a single market to determine the optimal allocation of the total system resource to each of the streams.
2. Use a variation of the First Fit Decreasing Utility heuristic to assign tasks to machines according to the optimal allocation.
3. Locally optimize the resource allocation to streams assigned to each machine.

Figure 4.3 illustrates how the system works. There are three roles in the system:

1. One ***controller*** for the entire system.
2. One ***machine agent*** for each machine.
3. One ***stream agent*** for each streaming application.

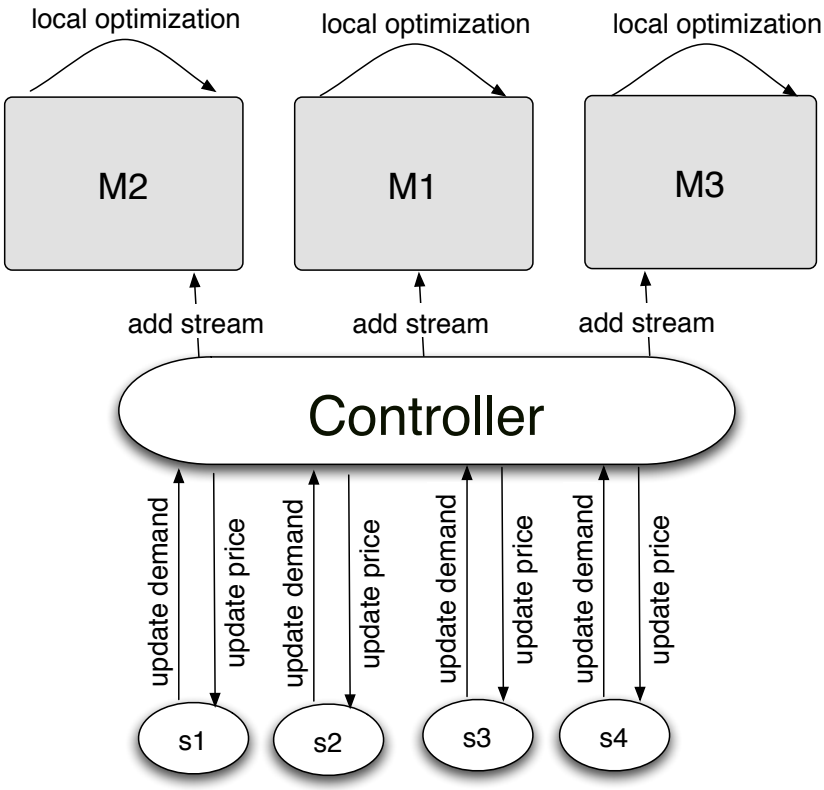


Figure 4.3: System view of the single-market method

4.1.1 Algorithm

The algorithm has the following three steps:

1. ***controller***: Assuming a virtual machine M_v with capacity equal to the sum of the capacities of the machines in the system $\sum C_i$, use the local optimization algorithm to solve for the optimal allocation $r_1 \dots r_n$.

2. ***controller***: Assign streaming applications to machines according to the *best-fit-decreasing-value* heuristic, where the size of each item is r_i and the value is $U(r_i)$ (details are presented below).

3. each ***machine agent***: Use the local optimization algorithm to determine the best allocation of each machine's resource to streaming applications.

best-fit-decreasing-value In this step, streaming applications are assigned to machines based on the optimal allocation found in the previous step. The heuristic we use is a modified version of the best-fit-decreasing-value heuristic commonly used in the bin-packing problem. Given the allocation $R = (r_1^* \dots r_N^*)$ and the corresponding utilities $U = (U_1(r_1) \dots U_N(r_N))$, do the following:

1. Sort the tasks in decreasing order of utility.

2. Apply the best-fit-decreasing-value heuristic to assign these sorted tasks to machines.

   (a) If the next task T_k can be assigned to a machine according to its optimal allocation r_k^*, do so.

   (b) Otherwise, if T_k can not be assigned to any machine according to the optimal allocation ($\forall j : r_k^* \geq remaining\_C_j$), put it in the set $\{RT\}$.

3. Assign the tasks in the set $\{RT\}$ to machines using the best-fit-decreasing-value heuristic, regardless of the fact that the sizes of streaming applications are greater than the remaining capacities on the machines.

4.1.2 Lower Bound Analysis

We prove that, in the worst case, the solution found by our heuristic is a 2-approximation.

Theorem 2 *Let U=$\sum U_i(r_i)$ be the total utility achieved by our solution and U_{opt} be the total utility achieved by the optimal allocation; then $U \geq \frac{1}{2}U_{opt}$.*

Proof Recall that our heuristic has three main steps: (1) run the local optimization algorithm on the virtual machine; (2) assign tasks to machines according to the resulting r_i; and (3) run the local optimization algorithm on each machine. The first step solves the underlying convex optimization problem exactly and finds the optimal allocation of resources to tasks assuming all machine resources are gathered into one virtual machine. Thus the total utility achieved on the first step, $\overline{U}$, serves as an upper bound on the optimal solution: $\overline{U} \geq U_{opt}$. If we can show that $U \geq \frac{1}{2}\overline{U}$, the proof is complete.

There are two substeps in the second step: (2a) assign tasks to machines according to r_i^* with the constraint $r_i^* \leq \mathit{remaining\_C_j}$; and (2b) assign remaining tasks to machines according to r_i^* without the constraint $r_i^* \leq \mathit{remaining\_C_j}$. We now prove that, after (2a), the sum of utilities of the assigned tasks is at least $\frac{1}{2}\overline{U}$.

Recall that each task has a value $U_i(r_i^*)$ and a size r_i^*. (2a) first arranges tasks in decreasing order of value $U_i(r_i^*)$, then assigns the largest-valued task to a machine unless it doesn't fit within the remaining capacity of any machine. Let $W_1 \ldots W_j$ be the wasted space on the machines after (2a). Let LT be the set of leftover tasks that can't be assigned to a machine according to their sizes r_i^*. Then

$$W = \sum W_j = \sum_{i \in \{LT\}} r_i.$$

We assumed that machine capacities were much larger than any individual task's resource requirement, thus

$$\max r_i^* < \min C_j.$$

When no remaining task can be assigned to any machine, we have:

$$\begin{aligned} \max W_j &< \min r_k^* \\ \overline{W_j} &< \overline{r_k} \\ \frac{W}{M} &< \frac{W}{|\,LT\,|} \\ |\,LT\,| &< M \end{aligned}$$

In the worst case, those M tasks are the ones with the $(M+1)st$, $(M+2)nd$, ..., $(2M)th$ highest values, and the sum of their values is less than or equal to the combined value of tasks $1 \ldots M$:

$$\sum_{i=M+1}^{2M} U_i(r_i^*) \leq \sum_{i=1}^{M} U_i(r_i^*)$$
$$\sum_{i=M+1}^{2M} U_i(r_i^*) \leq \frac{1}{2}\overline{U}.$$

Therefore, after (2a), the total utility realized by the tasks assigned to the machines is more than half of the optimal. It is apparent that the remaining steps never decrease the total utility: step (2b) adds remaining tasks to machines, and step (3) does a local optimization on each machine to re-balance the resource allocations to the tasks on that machine and maximize the total utilities. That is, if the original allocation (r_i^* to tasks assigned during (2a), and 0 to tasks assigned during (2b)) is optimal, then it is kept, otherwise the reallocation achieves higher total utility. Therefore, the total utility achieved by our heuristic is at least half of the optimal. □

From the proof, it is apparent that the $\frac{1}{2}$ lower bound is not tight. This conjecture is confirmed by experiment and simulation, as described in Chapter 5.

4.2 Multiple Market Resource Reservation System

The second resource reservation system we present models the system as multiple markets, one for each machine resource. This is analogous to the concept of multiple markets for substitutable goods in microeconomics theory. Consumers/streaming applications can choose to purchase goods/resources from different markets based on the price information available to them. Under the assumption that streaming applications can't split (each has to be pinned to one machine), the necessary condition for this M-commodity market to reach equilibrium is that the N streaming applications can be partitioned into M subsets, such that:

1. $U_i'(x_{ia}) = U_j'(x_{jb})$, for all $i, j \in [1, N]$
2. $Price_a = Price_b$, for all $a, b \in [1, M]$
3. $\sum_{i=1}^{N} x_{ia} = C_a$, for all $a \in [1, M]$

Under this formulation, market equilibrium is not guaranteed to exist. Furthermore, even if an equilibrium exists, the same lock-step algorithm used in the single-market heuristic doesn't converge to the equilibrium; instead, the system oscillates with applications repeatedly switching to lower priced machines. We therefore design a method to eliminate the oscillation.

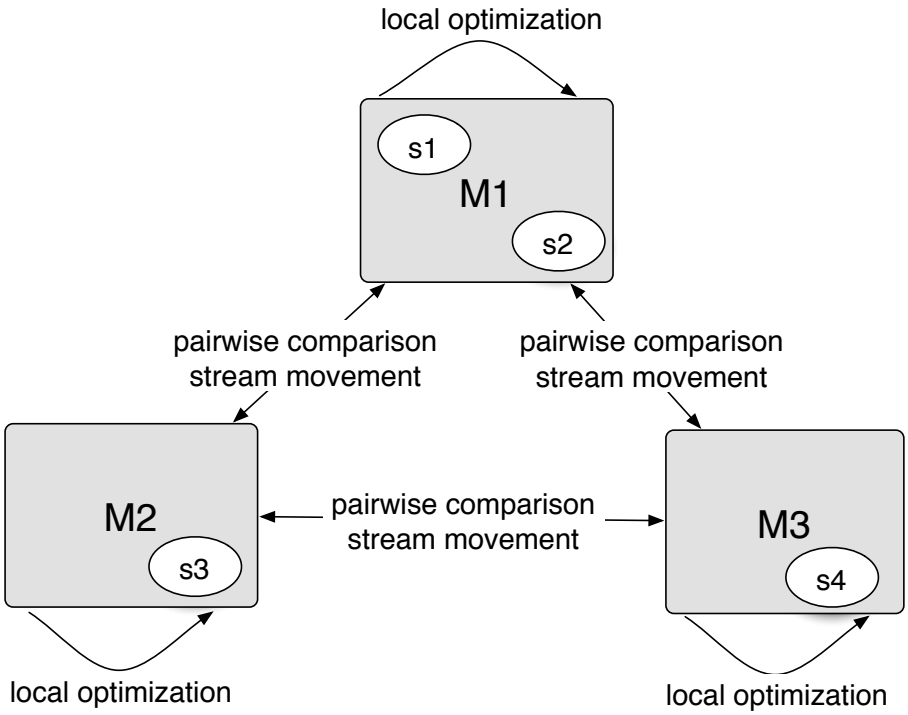


Figure 4.4: System view of the multiple-market method

Initially, each streaming application is assigned to one market (randomly or based on location). When all markets reach equilibrium, streaming applications from higher priced machines have incentives to move to lower priced machines. We enforce restrictions on when a streaming application may move from one machine to another: a move is allowed if and only if it increases the sum of the total utilities of the source and destination machines (their *pair-total utility*). Each step of this process involves a pair of machines attempting to move one streaming application from one to the other and comparing the pair-total utilities before and after this attempt. This is an iterative monotonic process that increases (or leaves constant) the total utility at each step and terminates when no streaming applications can be moved from one machine to another to increase the pair-total utility. Figure 4.4 illustrates how the system works. There are two roles in the system:

1. One ***machine agent*** for each machine.

2. One ***stream agent*** for each streaming application.

The algorithm is as follows:

1. Initialization: each streaming application is randomly assigned to a machine (*e.g.*, based on geographical location).

2. While there is at least one pair of streams in the system whose locations can be swapped to increase the total utility:

   (a) Each ***machine agent*** runs the local optimization algorithm to find the equilibrium price p_j^*.

   (b) ***machine agents*** communicate in pairwise fashion to move streams from one to another if the moves increase their pair-total utilities.

4.3 Summary

In this chapter, we have described two market-based heuristics for building resource reservation systems.

The single-market heuristic uses a single market to determine the optimal allocation of total resources to each streaming application, and then uses engineering heuristic NFDV to assign streaming applications to each machine based on the optimal allocation. It has provably polynomial running time and a $\frac{1}{2}$ lower bound on performance. We conjecture that this lower bound is loose (*i.e.*, the real performance is much better than this lower bound).

However, although most of the computation in the single-market heuristic is done by the individual stream agents and machine agents, all the price and demand information needs to be exchanged between the centralized controller and the agents. Therefore, the communications overhead is large and the system does not scale well.

The scaling issue motivated the development of the multiple-market heuristic. In this heuristic, the system is modeled as multiple markets, one for each machine. In addition to running the price adjustment algorithm to find the optimal resource allocation for each individual machine, machines are allowed to exchange streaming applications to increase the total utility. The multiple-market heuristic is completely decentralized and scales well to large numbers of machines and streams; however, there is no theoretical complexity bound on its performance.

In the next chapter, we will use simulations to study the performances of these two methods.

Chapter 5

Simulations

We evaluate the performances of the two heuristics developed in Chapter 4 through simulation. In the experiments, we assume that all streaming applications have utility functions of the form $U_i(r_i) = b_i \frac{r_i^{(1-a_i)}}{(1-a_i)}$, $a_i \in (0,1)$, $b_i \in (0,\infty)$. We choose this particular class of functions because it captures/approximates many concave functions that are typically used as utility functions [25]. To evaluate performance, we compare the total utility achieved by our heuristics with two metrics:

1. **Upper bound $\overline{\text{U}}$**: the maximum total utility achieved assuming streams can split.
2. **Base bound $\underline{\text{U}}$**: the total utility achieved by the *naïve balanced-streams* heuristic, which assigns streams to balance the number of streams per machine.

In most of the figures below, we compare the performances of our heuristics and the balanced-streams heuristic by plotting the *normalized performance gap (NPG)* for each. The NPG for a heuristic is calculated as $\frac{\overline{U}-U}{\overline{U}}$, where U is the total utility achieved by the heuristic and $\overline{U}$ is the performance upper bound. Thus, the NPG is a number between 0 and 1; the smaller the NPG, the better the performance.

5.1 A Motivating Example

The balanced-streams heuristic has a randomness factor in its performance. That is, for N streaming applications and M machines, it has M^N possible assignments, each occurring with equal probability but resulting in different performance. We first use a motivating example to show that, in some cases, the naïve approach achieves arbitrarily bad performance where our heuristics each achieve near-optimal performance.

Suppose there are two machines, each with unit capacity $C_1 = C_2 = 1$, and four streaming applications $T_1 \ldots T_4$, each with utility function $U_i(r_i) = b_i \frac{r_i^{(1-a_i)}}{(1-a_i)}$, where $a_1 = a_2 = 0.1$, $b_1 =$

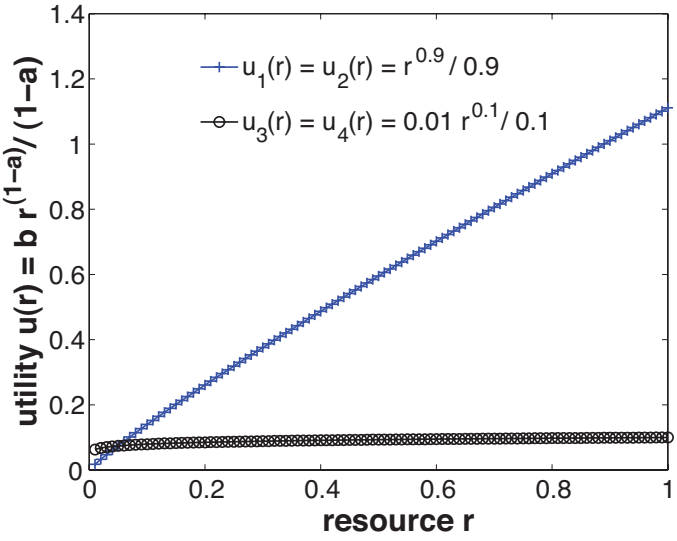


Figure 5.1: Utility functions for $T_1 \ldots T_4$

$b_2 = 1$, $a_3 = a_4 = 0.9$, and $b_3 = b_4 = 0.01$. Thus $U_1(r_1) = U_2(r_2) = \frac{r^{0.9}}{0.9}$ and $U_3(r_3) = U_4(r_4) = 0.01\frac{r^{0.1}}{0.1}$ as shown in Figure 5.1.

The balanced-streams heuristic has six possible assignments with equal probabilities. It has $\frac{1}{3}$ probability of assigning T_1 and T_2 to the same machine, which results in allocations of $r_1 = r_2 = r_3 = r_4 = 0.5$ and total utility of $\underline{U} = 0.5953 + 0.5953 + 0.0933 + 0.0933 = 1.3772 = 0.59U_{opt}$. Our market-based heuristics always assign T_1 and T_2 to different machines, resulting in allocations of $r_1 = r_2 = 0.994$ and $r_3 = r_4 = 0.006$ and total utility of $U = 1.105 + 1.105 + 0.06 + 0.06 = 2.33 = U_{opt}$. Thus, the total utility achieved by the naïve balanced-streams heuristic is only $\frac{1.3772}{2.33} = 59\%$ of that achieved by our market-based heuristics for this example setup.

Next, we compare the metrics under two distributions: a *heavy-tail distribution* and a *uniform distribution*.

5.2 Heavy-tail Distribution

The stream processing applications follow a heavy-tail distribution if, under the optimal allocation on the virtual machine, many streams have small allocations, a few streams have large allocations, and very few fall in between. This distribution is typical for flows on the Internet, and is known as the "elephants and mice" distribution.

To test the performances of our heuristics under this distribution, we run the following experiments. Assume there are 2 machines in the system, each with unit capacity, and $N = \{3,4,5,6,7,8,9\}$ streaming applications each with a utility function $U_i(r_i) = b_i \frac{r_i^{(1-a_i)}}{(1-a_i)}$. Let M of the streaming applications have the parameters $a = 0.1$, $b = 1$, and $N - M$ have the parameters $a = 0.9$, $b = 0.01$.

Under this experimental setup, the balanced-streams heuristic has M^N different assignments. To compare the performances of the heuristics, we compute the average performance of the balanced-streams heuristic. This is done by brute force:

1. Run the heuristic under each of the possible assignments.

2. Compute the average utilities achieved by all possible assignments.

As shown in Figure 5.2, the average performance of the balanced-streams heuristic stays at 85% of optimal, while both of our heuristics perform near 100% of optimal. The fully-decentralized multiple-market heuristic performs just as well as the single-market heuristic.

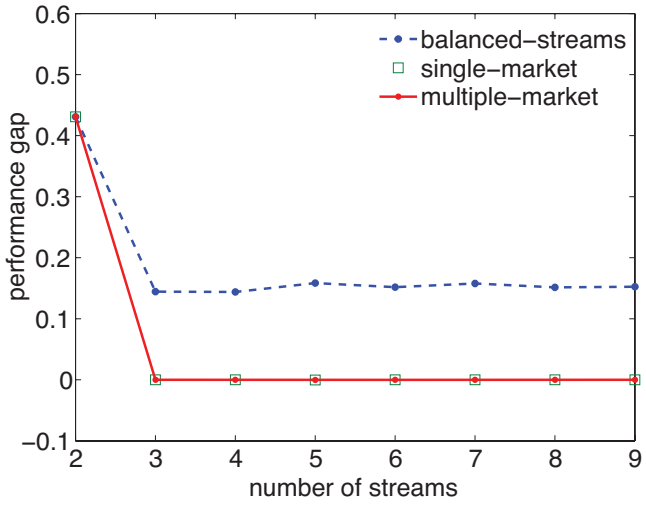


Figure 5.2: Performances of the three heuristics under a heavy-tail stream distribution

5.3 Uniform Distribution

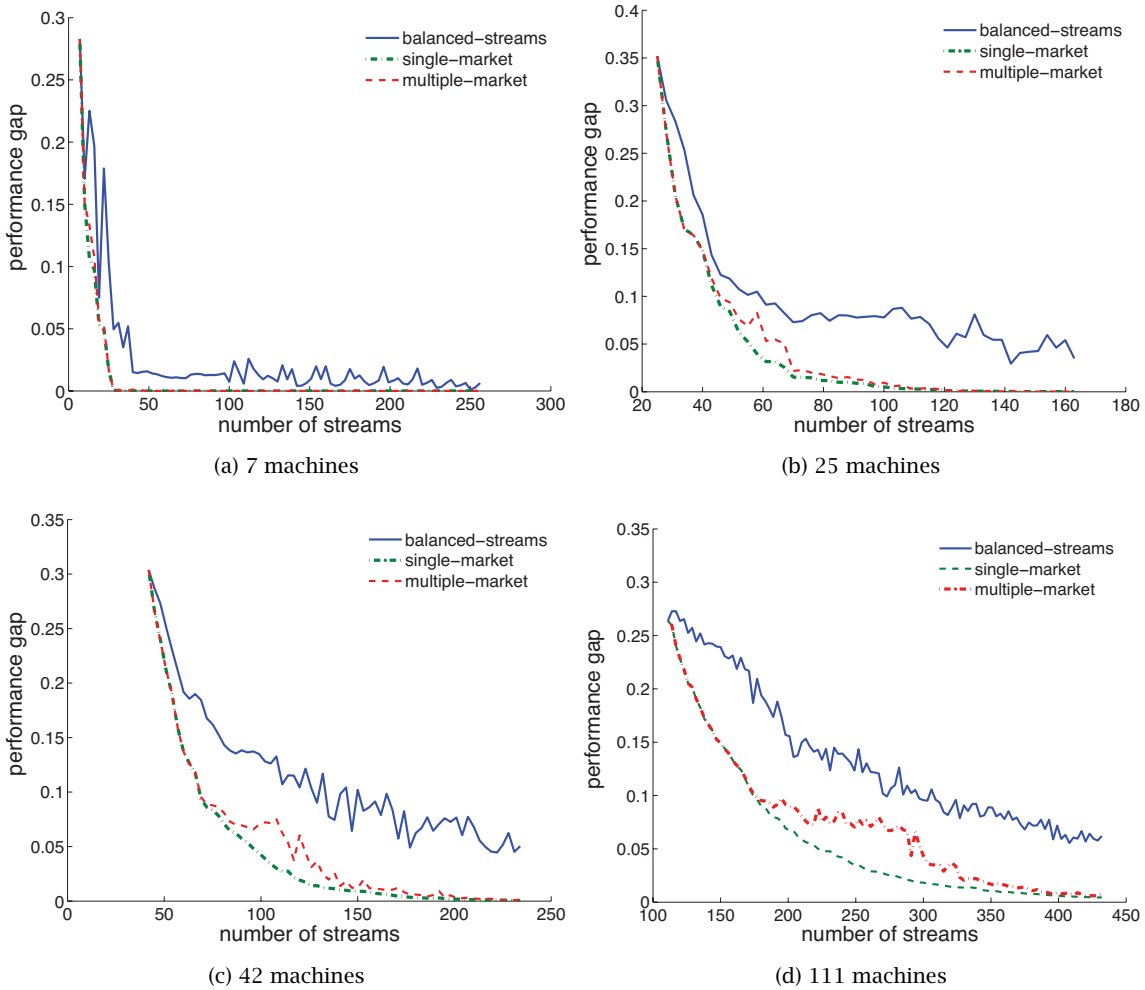
In this section, we analyze how different factors affect the performances of the three heuristics when the streaming applications follow a uniform distribution.

5.3.1 Comparison Among the Three Heuristics

To compare the performances of the three heuristics under a uniform stream distribution, we use the following experimental setup:

Generate N streaming applications, each with a utility function $U_i(r_i) = b_i \frac{r_i^{(1-a_i)}}{(1-a_i)}$, with $a_i \in (0.2, 0.3)$ and $b_i \in (0, 1)$ chosen randomly. Generate M machines, each with capacity 1.

Figure 5.3 shows the performances of the three heuristics with 7, 25, 42 and 111 machines in the system.



(a) 7 machines

(b) 25 machines

(c) 42 machines

(d) 111 machines

Figure 5.3: Performances of the three heuristics under a uniform stream distribution, with various numbers of machines

The following observations are based on the simulation results:

1. The performances of our two market-based heuristics approach optimal quickly as the number of streaming applications increases.

2. The balanced-streams heuristic performs significantly worse than our market-based heuristics in all cases.

3. The performance gap for the balanced-streams heuristic decreases as the number of streams increases, but doesn't converge to optimal (for up to 500 streams in the system), staying from 1% to 10% below optimal.

4. The more machines in the system, the larger the performance gap for the balanced-streams heuristic, and the slower the convergence of the two market-based heuristics.

5.3.2 The Effect of Number of Machines on Performance

To see how the number of machines affects the performance for each heuristic, we generate one plot for each heuristic showing the performances of that heuristic with various numbers of machines.

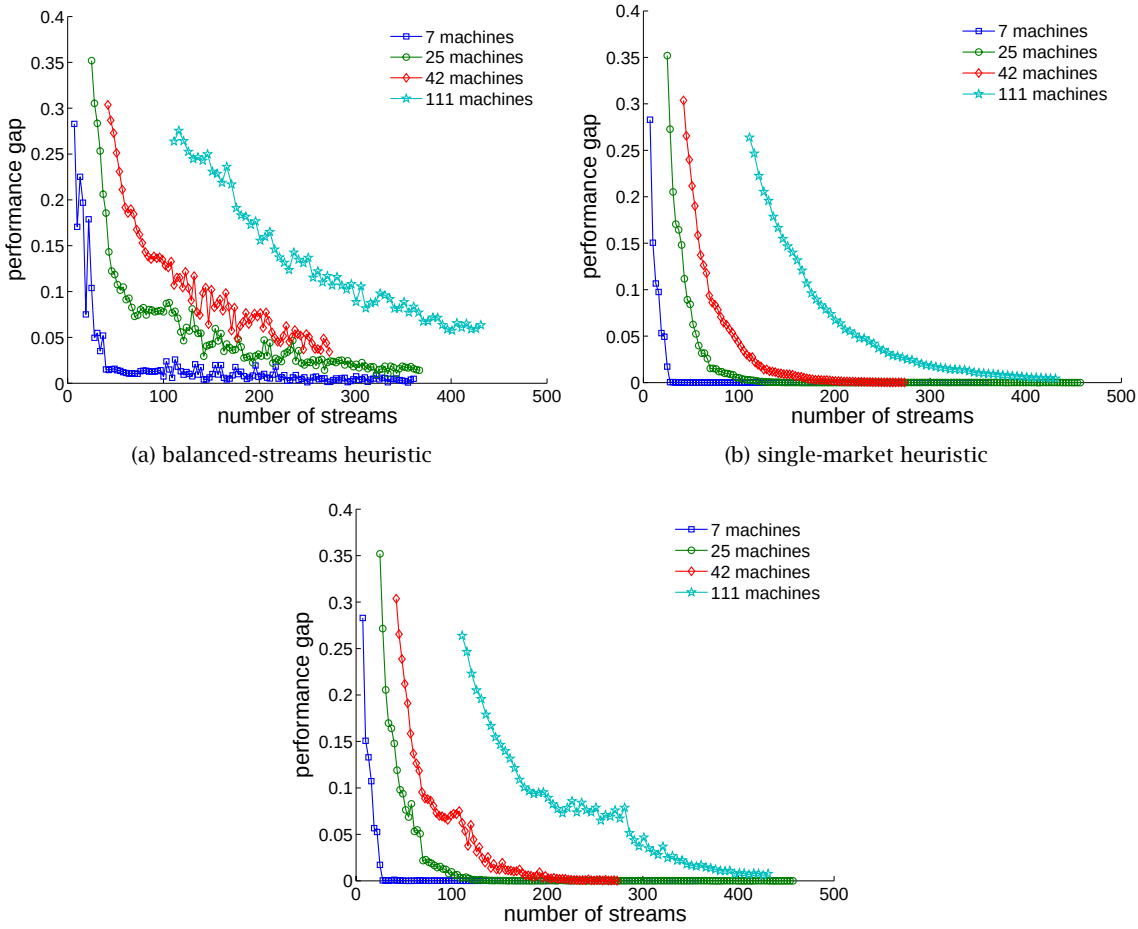
We use the same setup as the one in the previous section: N streaming applications, each with a utility function $U_i(r_i) = b_i \frac{r_i^{(1-a_i)}}{(1-a_i)}$, with $a_i \in (0.2, 0.3)$ and $b_i \in (0, 1)$ chosen randomly; M machines, each with capacity 1. The following observations are based on the simulation results shown in Figure 5.4:

1. For each fixed number of streaming applications in the system, the performance gap for the naïve balanced-streams heuristic increases as the number of machines increases.

2. Initially, for each fixed number of streaming applications in the system, the performance gaps for the two market-based heuristics increase as the number of machines increases; however, with increasing numbers of streaming applications, the performance gaps for all heuristics quickly converge to zero.

3. The fully-decentralized multiple-market heuristic performs almost as well as the single-market heuristic.

Recall that the upper bound is calculated by aggregating all machine resources into a virtual machine and allocating the aggregated resource to streams. Thus, each streaming application gets an 'optimal' amount of resource (the more competitive streams get more resource, and the less competitive streams get less resource). These optimal allocations result in a maximized total utility. Furthermore, the performance gap is calculated as $\frac{\overline{U}-U}{\overline{U}}$, where U is the total utility achieved by the heuristic and $\overline{U}$ is the performance upper bound.



(a) balanced-streams heuristic

(b) single-market heuristic

(c) multiple-market heuristic

Figure 5.4: The effect of number of machines on heuristic performance

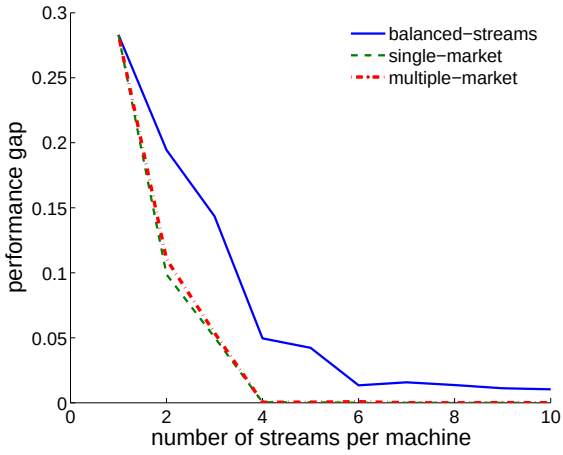
The value $\overline{U} - U$ has two components:

1. The gap between the upper bound and the optimal value.

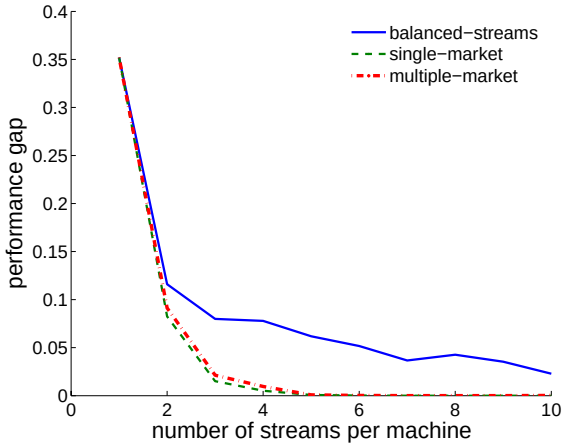
2. The gap between the optimal value and the heuristic result.

The first component is called the *duality gap*. It is the value forgone because of the 'indivisible' nature of the problem, and is irrelevant to the goodness of the heuristic.

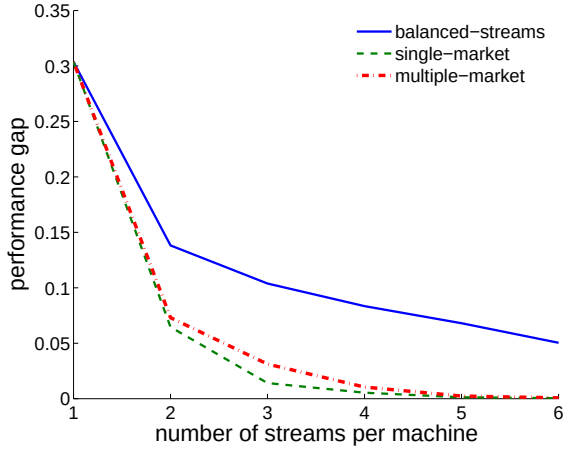
One reason that the performance gap increases with increasing number of machines is a larger duality gap; when streams are assigned to machines, the 'optimal' allocations can't be preserved because the total resource is fragmented among the machines. When there are more machines in the system, the total resource is more fragmented, so more streams are unable to get their 'optimal' allocations. This results in a larger duality gap between the upper bound and what can possibly be achieved.



(a) 7 machines



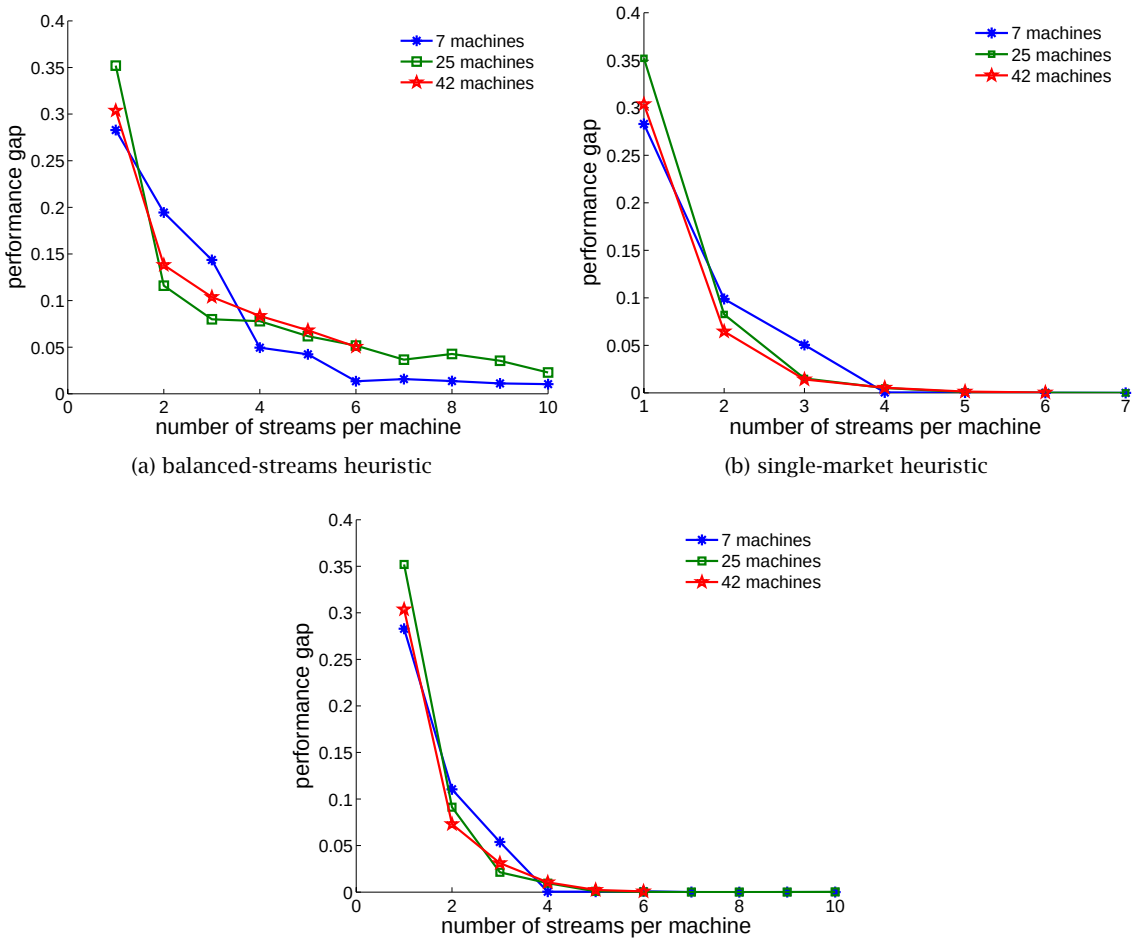
(b) 25 machines



(c) 42 machines

Figure 5.5: The effect of streams-to-machine-ratio on performance, by number of machines

To analyze how the relative number of streams to machines affects the performance gap, we plot the performance gaps of the three heuristics with respect to the number of streams per machine. Figures 5.5 and 5.6 show the comparisons of performance gaps among the three heuristics for various numbers of machines. As we can see from these plots, in all cases, the performance gaps of our market-based heuristics converge to 0 when there are more than 4 streams per machine while the naïve balanced-streams heuristic stays at a 1% to 4% performance gap (depending on the number of machines).



(a) balanced-streams heuristic

(b) single-market heuristic

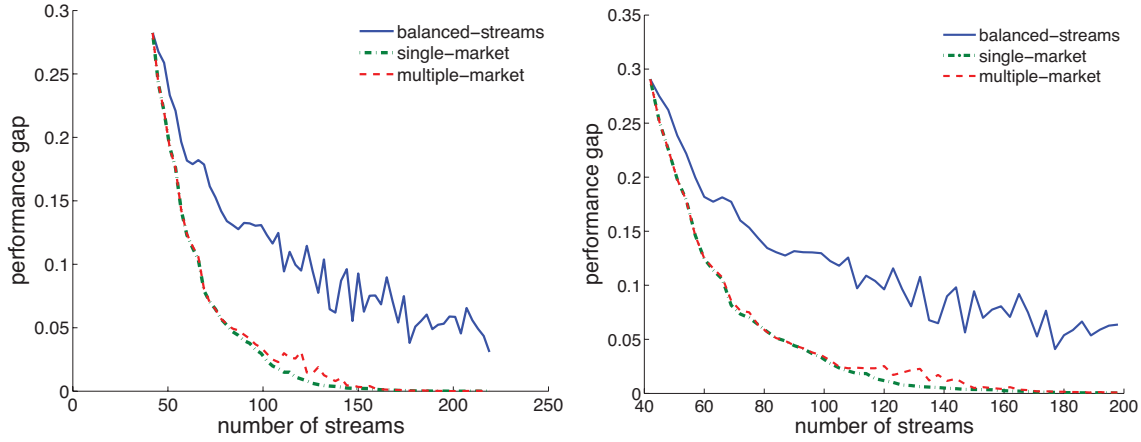
(c) multiple-market heuristic

Figure 5.6: The effect of streams-to-machine-ratio on performance, by heuristic

5.3.3 The Effect of Total System Resource on Performance

To study how the total system resource affects the performance, we run the heuristics with three different mean values for machine capacity: 1, 0.1, and 0.01. We use a setup similar to the one in the previous section: N streaming applications, each with a utility function $U_i(r_i) = b_i \frac{r_i^{(1-a_i)}}{(1-a_i)}$, with $a_i \in (0.2, 0.3)$ and $b_i \in (0, 1)$ chosen randomly.

Each plot in Figure 5.7 shows the performances of the three heuristics with a distinct CPU mean value in a system with 42 machines. Figure 5.8 has one plot per heuristic, which shows the heuristic's performance under three CPU mean values with 42 machines in the system.



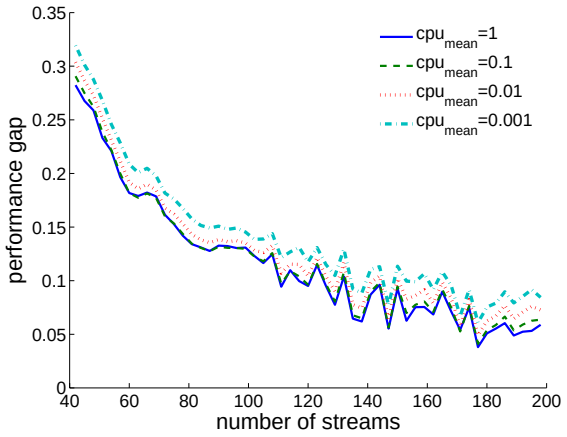
(a) cpu mean = 1

(b) cpu mean = 0.1

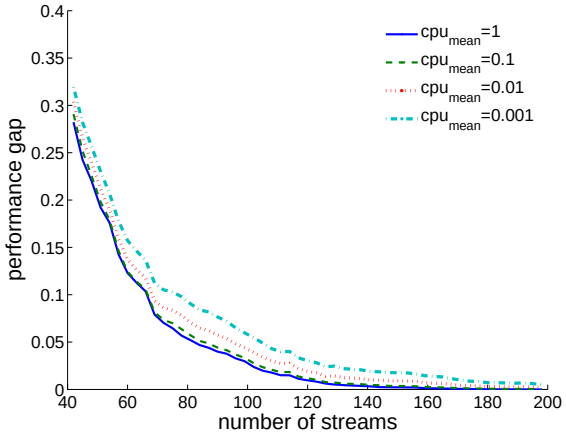
(c) cpu mean = 0.01

(d) cpu mean = 0.001

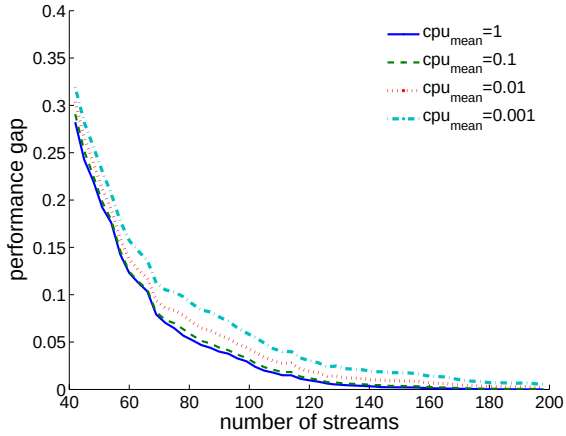
Figure 5.7: The effect of total system resource on performance, by capacity (42 machines)



(a) balanced-streams heuristic



(b) single-market heuristic



(c) multiple-market heuristic

Figure 5.8: The effect of total system resource on performance, by heuristic (42 machines)

The following observations are based on these plots:

1. **Balanced-streams heuristic**:
   (a) The performance gap is larger when the total system resource is smaller.
   (b) The performance gap decreases as the number of streams increases, but never converges.
2. **Single-market heuristic**:
   (a) The performance gap is larger when the total system resource is smaller.
   (b) The performance gap decreases as the number of streams increases, and eventually converges to zero. The smaller the total resource, the slower it converges.

3. **Multiple-market heuristic**:

   (a) The performance gap is larger when the total resource is smaller.

   (b) The performance gap decreases as the number of streams increases and eventually converges to zero. The smaller the total resource, the slower it converges.

   (c) Performance is almost as good as that of the single-market heuristic.

One reason that the performance gap increases with decreasing amount of total resource is the larger duality gap. An intuitive explanation is: when the total resource is scarce (supply is low), while the demands from the streams remain the same, the market is more competitive. Under a more competitive market, it is more important that the streams get their 'optimal' allocations because a small perturbation will result in a large utility loss. The theory behind this intuition is that, when the total resource is less, the resource each stream gets is also less. Since each stream has a concave utility function with decreasing marginal utility with respect to resource, less resource results in larger marginal utility. Thus, the system is more sensitive to the fact that the resource is fragmented among a number of machines. Therefore, a smaller amount of total resource results in a larger marginal utility, which in turn results in a larger performance gap.

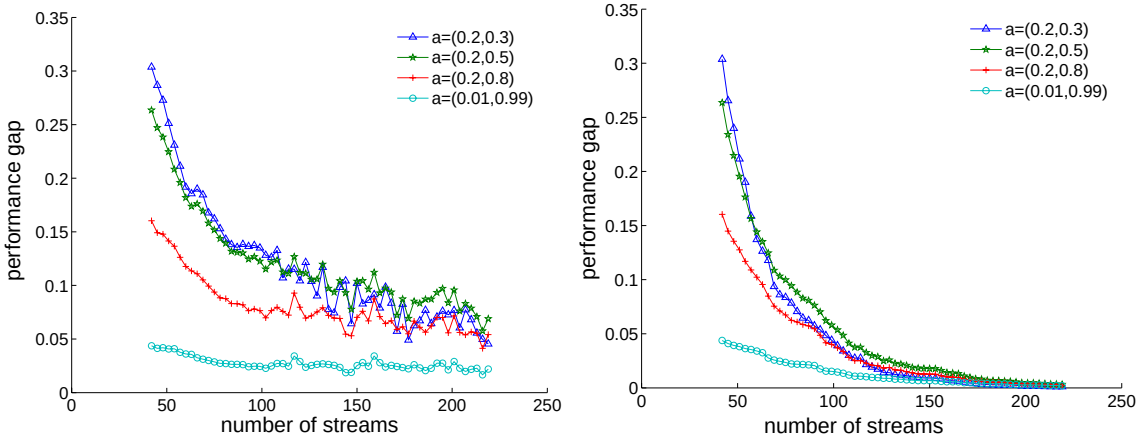
The decrease in performance gap with increasing number of streaming applications is due to the decreasing duality gap; the more streaming applications in the system, the less resource each of them gets under the upper bound calculation. Thus, fewer of them fail to get their 'optimal allocation' when they are assigned to individual machines. This is similar to the bin packing example; given a set of fixed bins, it is easier to assign smaller balls to the bins than larger balls, so less bin volume is wasted by an inability to assign balls to bins.

5.3.4 The Effect of Utility Functions on Performance

Recall that each streaming application has a utility function of the form $U_i(r_i) = b_i \frac{r_i^{(1-a_i)}}{(1-a_i)}$, $a_i \in (0,1)$, $b_i \in (0,\infty)$. In this section, we study how the choices of a and b affect the performances of the three heuristics.

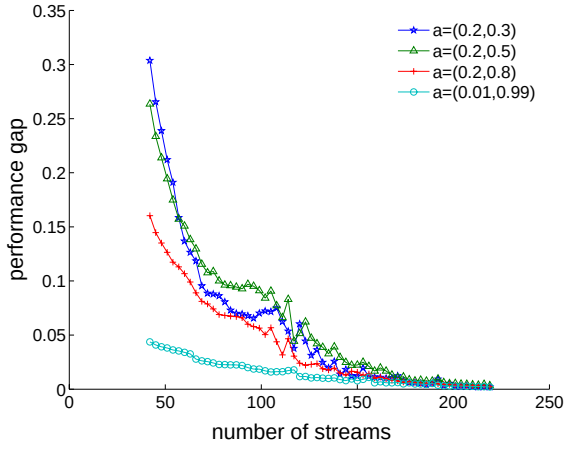
5.3.4.1 Choice of a

To test how parameter a affects performance, we fix the range of b to be $(0,1)$ and the machine capacity to be 1, and run the heuristics with various ranges of a: (1) $a \in (0.2, 0.3)$; (2) $a \in (0.2, 0.5)$; (3) $a \in (0.2, 0.8)$; and (4) $a \in (0.01, 0.99)$. As shown in Figures 5.9 and 5.10, when the mean of a is smaller, the performance gap tends to be larger.



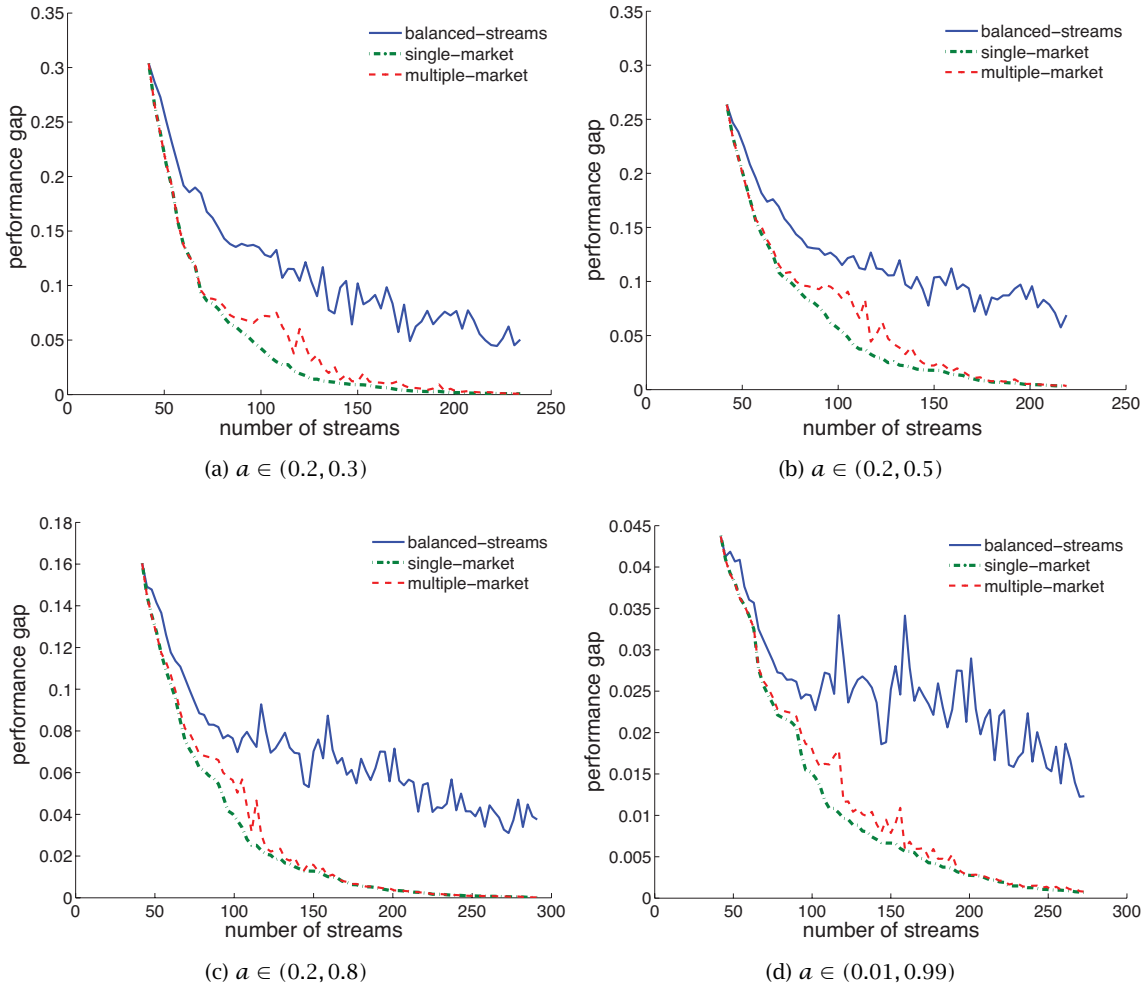
(a) balanced-streams heuristic

(b) single-market heuristic



(c) multiple-market heuristic

Figure 5.9: Performance comparison with various ranges of a, by heuristic (42 machines)



(a) $a \in (0.2, 0.3)$

(b) $a \in (0.2, 0.5)$

(c) $a \in (0.2, 0.8)$

(d) $a \in (0.01, 0.99)$

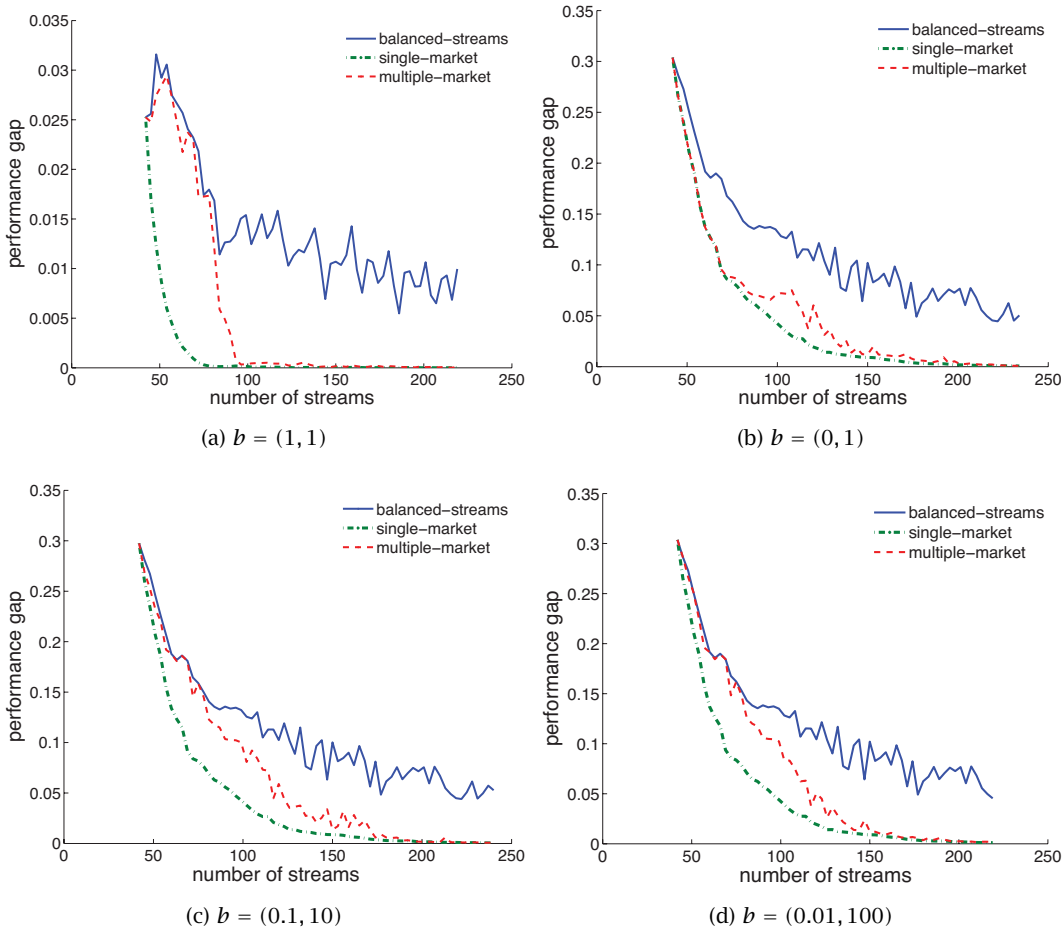
Figure 5.10: Performance comparison with various ranges of a, by range (42 machines)

Recall that each stream has a concave utility function of the form $U_i(r_i) = b_i \frac{r_i^{1-a_i}}{1-a_i}$. Thus the marginal utility is $U_i'(r_i) = b_i r^{-a_i}$. When a is smaller, the marginal utility becomes larger. As discussed earlier, larger marginal utilities cause larger performance gaps. This explains why the performance gaps are larger with smaller a.

However, there is a contradiction: when there are more than 70 streams in the system, $a \in (0.2, 0.5)$ has a higher performance gap than $a \in (0.2, 0.3)$. This is because, as the variance of a becomes larger, the streams become more diverse and a *complementary effect* takes over. This complementary effect occurs because it is easier to fill the machines with streams of diverse sizes than to fill them with equally sized streams. As an analogy, consider the problem of filling bins with balls. It is easier to fill the bins with balls of different sizes than with balls of same size. When there is a large number of streams in the system, the complementary effect is more influential and subsumes the a mean effect.

5.3.4.2 Choice of b

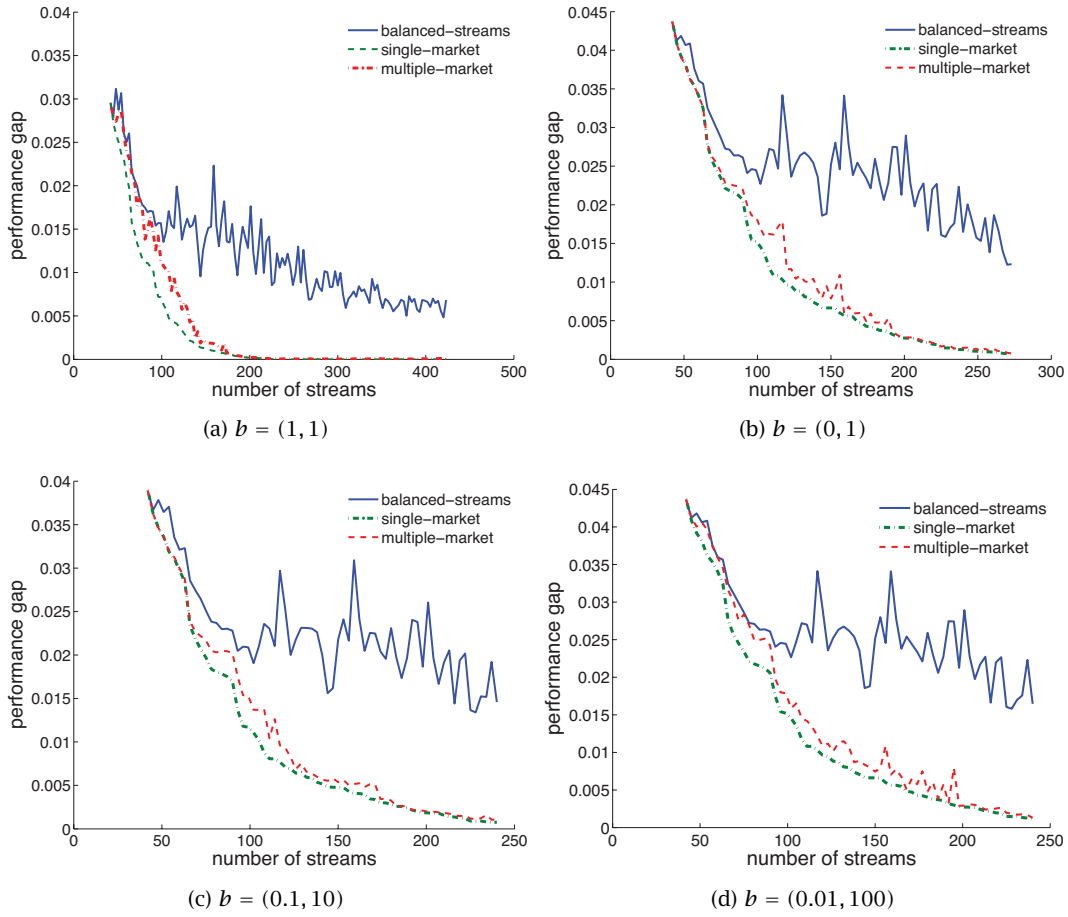
To test how parameter b affects performance, we fix the range of a and run the heuristics with various ranges of b: (1) $b \in (1,1)$; (2) $b \in (0,1)$; (3) $b \in (0.1,10)$; and (4) $b \in (0.01,100)$. As shown in Figures 5.11 and 5.12, regardless of the range of b, the two market-based heuristics perform significantly better than the naïve balanced-streams heuristic.



(a) $b = (1,1)$ (b) $b = (0,1)$

(c) $b = (0.1,10)$ (d) $b = (0.01,100)$

Figure 5.11: Performance comparison with various ranges of b, by range (42 machines, $a \in (0.2,0.3)$)

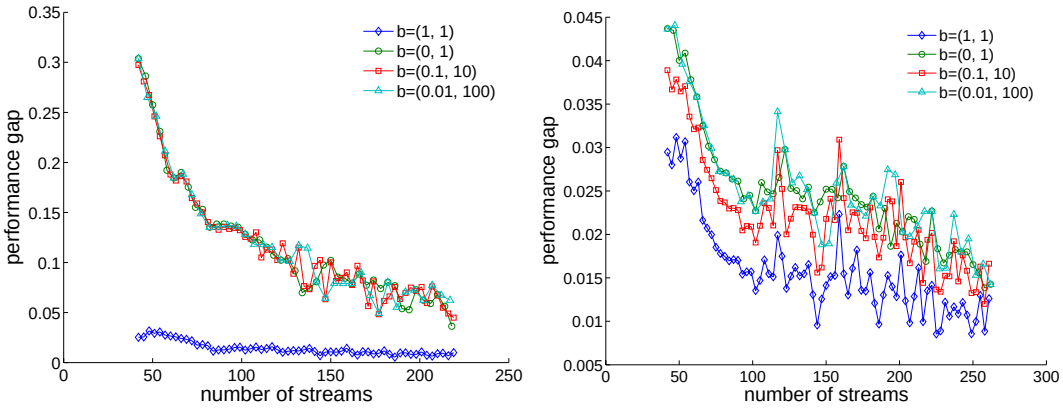


(a) $b = (1, 1)$ (b) $b = (0, 1)$

(c) $b = (0.1, 10)$ (d) $b = (0.01, 100)$

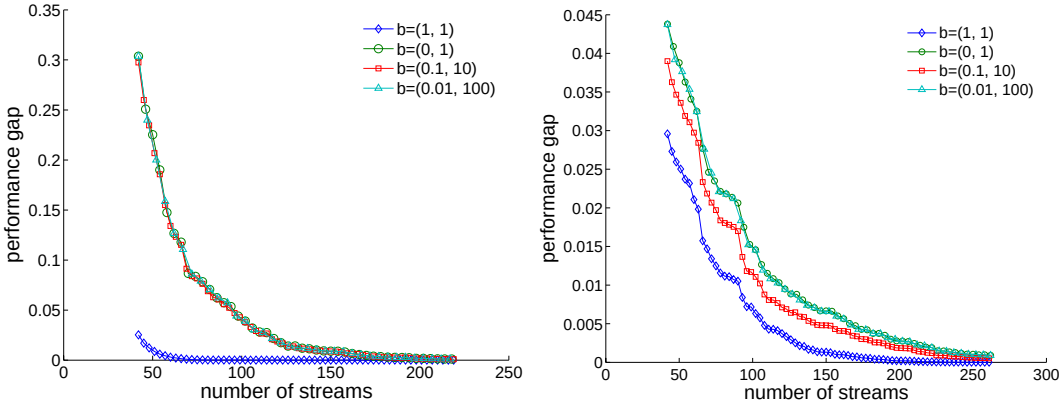
Figure 5.12: Performance comparison with various ranges of b, by range (42 machines, $a \in (0.01, 0.99)$)

From Figure 5.13 it can be observed that, with fixed CPU mean of 1 and for two tested ranges of a, the performance gap increases with the mean of b. This occurs for a reason similar to that discussed previously for a. Since $U_i'(x) = bx^{-a}$, U_i' is larger when b is larger; therefore, the performance gap is larger. This behavior can be best observed in Figures 5.13(b)(d)(f), where $a \in (0.01, 0.99)$. We discussed the complementary effect earlier when analyzing how the choice of a affects the performance. This complementary effect can also be induced by the choice of b; the larger the variance in b, the larger the complementary effect. This explains why the marginal utility behavior is more apparent when $a \in (0.01, 0.99)$ than when $a \in (0.2, 0.3)$; when $a \in (0.01, 0.99)$, the streams are already more diverse than when $a \in (0.2, 0.3)$, so increasing b to increase diversity has little impact.



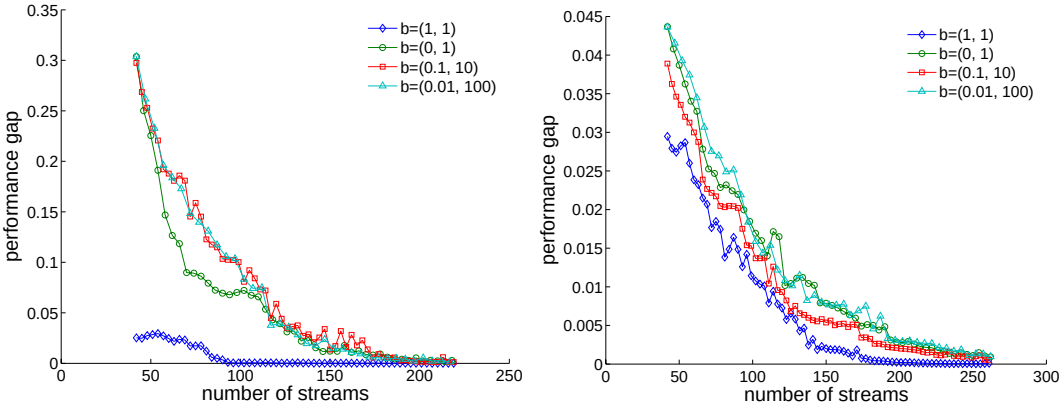
(a) balanced-streams heuristic, $a \in (0.2, 0.3)$

(b) balanced-streams heuristic, $a \in (0.01, 0.99)$



(c) single-market heuristic, $a \in (0.2, 0.3)$

(d) single-market heuristic, $a \in (0.01, 0.99)$



(e) multiple-market heuristic, $a \in (0.2, 0.3)$

(f) multiple-market heuristic, $a \in (0.01, 0.99)$

Figure 5.13: Performance comparison with various ranges of b and two ranges of a, by heuristic (42 machines)

5.3.5 Timing Analysis for the Multiple-Market Heuristic

The single-market heuristic has provably polynomial running time. However, there is no theoretical complexity bound on the performance of the multiple-market heuristic. We study the timing/computational complexity of the multiple-market heuristic empirically by measuring how long it takes for the process to converge during each simulation. We measure time by the number of rounds it takes for each simulation to complete, where each round is one attempt to move a streaming application from one machine to another (each attempt essentially entails running the local optimization step once). The average running time for the local optimization step is 10 ms.

We plot the number of rounds against the number of streams in the system. Figure 5.14 shows four curves, one for each distinct number of machines in the system. One observation is that the running time increases with the number of streams in the system but does not depend on the number of machines in the system.

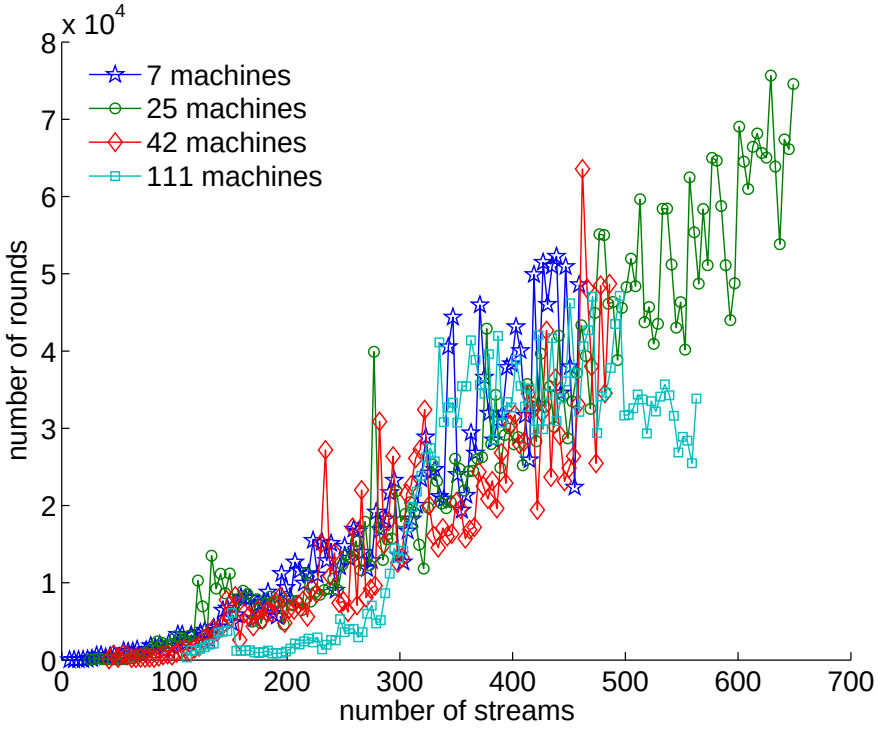


Figure 5.14: Timing comparison

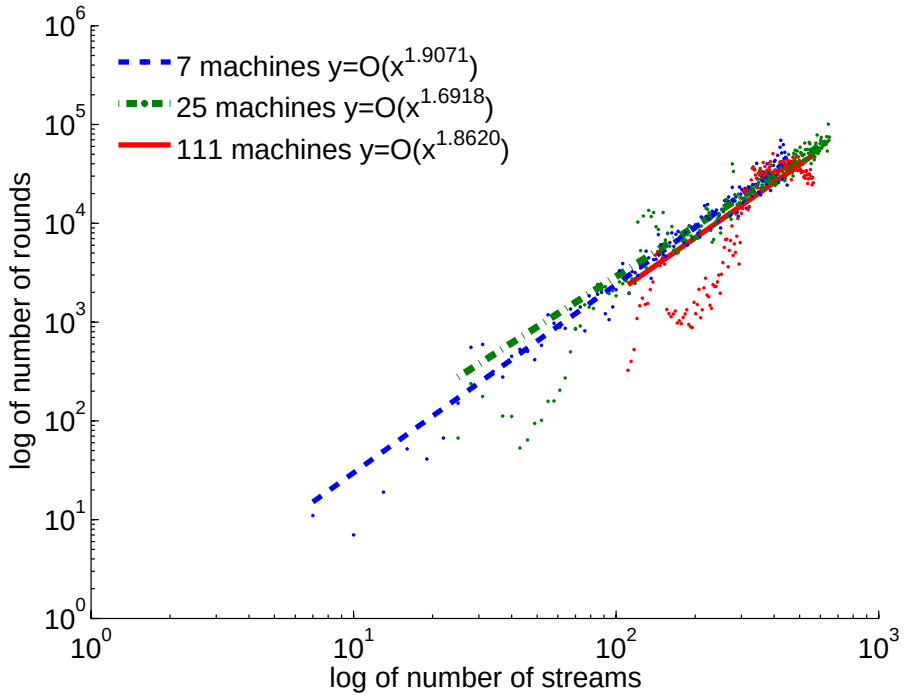


Figure 5.15: Timing approximation

To study how the execution time changes with increasing number of streams in the system, we attempt to approximate it with a polynomial function. Suppose this function is of order x: $T(n) = O(n^x)$, where n is the number of streaming applications. We derive the following:

$$\log(T(n)) = \log(n^x)$$
$$\log(T(n)) = x\log(n)$$
$$x = \frac{\log(T(n))}{\log(n)}$$

Thus if we plot $\log(time)$ *vs.* $\log(number\_of\_streams)$ for each distinct number of machines, x can be approximated by the slope of this log-log plot. Such a plot for 7, 25 and 111 machines appears in Figure 5.15.

It is evident from the empirical result that the time complexity of the multiple-market heuristic is approximately quadratic in the number of streams.

Chapter 6

Conclusion

6.1 Summary

In this thesis, we have described the emerging class of streaming applications and addressed the fact that the existing model and method for solving scheduling problems are not suitable for streaming applications because of their differences from conventional tasks. We introduced a new performance model and solution space (resource reservation system) that are better suited for modeling and solving the scheduling/resource allocation problem in the streaming environment. We also proposed two market-based heuristics for building such resource reservation systems. The single-market heuristic is semi-decentralized with provably polynomial running time. The multiple-market heuristic is fully decentralized with empirically polynomial running time. We proved a $\frac{1}{2}$ lower bound on the performance of the semi-decentralized heuristic.

We then evaluated the performances of these two heuristics by comparing them with that of a naïve load balancing heuristic, balanced-streams. Experimental results show that the balanced-streams heuristic performs better under a uniform distribution than under a heavy-tail distribution. Given a fixed number of machines in the system, the performance gap decreases as the number of streams increases, however, it never converges to optimal but stays around 90% of optimal. This is still significantly worse than our single-market heuristic.

Our polynomial time single-market heuristic performs well on both uniform and heavy-tail distributions, and outperforms the naïve balanced-streams heuristic significantly on the heavy-tail distribution. Given a fixed number of machines in the system, the performance gap decreases as the number of streams increases and converges to optimal quickly. However, this heuristic is based on a single market: although computations are distributed, all price and demand information flows to a centralized scheduler. Thus, this scheme does not scale to large systems that have no centralized locus of information. We therefore devised another heuristic that is fully decentralized.

It is evident that, under both types of stream distributions, the fully-decentralized multiple-

market heuristic performs almost as well as the single-market heuristic without requiring a centralized market/scheduler. This makes it a good candidate for use in large distributed systems and grid-like environments.

However, our multiple-market heuristic is not provably polynomial in time complexity. To determine the complexity of this heuristic, we did an empirical study by measuring the time it takes to finish in simulation. We then analyzed the complexity by plotting the logarithm of time *vs.* the logarithm of the number of streams for various numbers of machines to see how the system scales as the number of streams increases. The results show that this heuristic's running time is independent of the number of machines in the system, and is approximately quadratic in the number of streaming applications. Thus it scales well with increasing numbers of streams and machines, which makes it a practical and attractive alternative to the single-market heuristic.

6.2 Applications

Significant work has been done in the area of applying economic principles to resource allocation in grids. Some of this work is based on auctions [29], while some is based on competitive markets [2]. Wolski *et al.* [31] provide a good comparison of these projects. Research that integrates computer science with economics has also proved useful with respect to the Internet [26, 28].

Our resource reservation system designs were carried out using concepts from the literature on scheduling and economics. The market-based heuristics presented in this thesis for building resource reservation systems can be used for general resource allocation on heterogenous computing systems for streaming applications, and have a wide range of applications. Examples of possible application areas are the following:

Computational Grids Grid computing "is an emerging computing model that provides the ability to perform higher throughput computing by taking advantage of many networked computers to model a virtual computer architecture that is able to distribute process execution across a parallel infrastructure." [30] Currently, computational grids focus on providing support for massive computations such as protein folding, financial modeling, earthquake simulation, and climate/weather modeling. We believe that computational grids can also be used as the computational fabric for processing streaming applications. Thus, making the best possible use of resources on computational grids to process these streams and generate the most economic value becomes crucially important. Our market-based heuristics, especially the fully-decentralized multiple-market heuristic, are good candidates for resource management

for streaming applications on computational grids.

Enterprise Computing As sensors and RFID tags become more widely deployed, and streams of data from commodity exchanges, wire services, and other sources become more widely and freely available, new application areas for enterprise computing systems are emerging. It is becoming increasingly important for enterprises to build applications that use this data to detect and react to potential threats and opportunities ("critical states").

Streaming applications analyze events from many different sources and of many different forms—numerical, textual, and visual—to determine when a critical state exists and what the appropriate response should be. They allow enterprise computing systems to act as "information factories"; just as industrial factories create value by transforming raw materials into finished products, information factories create value by transforming raw events into structured data. This transformation can take considerable computational resources, so it is important both to design efficient and reliable heuristics and to make the best possible use of the available computing infrastructure when executing these heuristics.

6.3 Future Directions

We conjecture that the $\frac{1}{2}$ lower bound for the single-market heuristic is not tight. This has been confirmed by simulation results. We are trying to formulate a proof that tightens this bound. We are also refining optimization techniques for the multiple-market heuristic.

In addition, we are working on optimization techniques for the multiple-market heuristic. We also plan to study various extensions of this problem by relaxing existing assumptions. For example, when the stream processing applications do not have the same existence interval, the problem becomes a stochastic problem; when the stream processing applications consist of interacting graphs of processing units, communication costs need to be taken into account when making scheduling decisions.

Bibliography

[1] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çentintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the Borealis stream processing engine. In *Conference on Innovative Data Systems Research*, 2005. 14

[2] D. Abramson, R. Buyya, and J. Giddy. A computational economy for grid computing and its implmentation in the Nimrod-G resource broker, 2002. 50

[3] S. Ali, J.-K. Kim, Y. Yu, S. Gundala, S. Gertphol, H. Siegel, A. Maciejewski, and V. Prasanna. Greedy heuristics for resource allocation in dynamic distributed real-time heterogeneous computing systems. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, June 2002. 1, 4

[4] R. Armstrong, D. Hensgen, and T. Kidd. The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions. In *7th IEEE Heterogeneous Computing Workshop (HCW '98)*, March 1998. 5

[5] S. Babu and J. Widom. Continuous queries over data streams. In *SIGMOD Record*, September 2001. 14

[6] Tracy D. Braun, Howard Jay Siegel, Noah Beck, Ladislau L. Bölöni, Muthucumaru Maheswaran, Albert I. Reuther, James P. Robertson, Mitchell D. Theys, Bin Yao, Debra Hensgen, and Richard F. Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61(6):810–837, 2001. 1

[7] Almut Burchard, Jorg Liebeherr, Yingfeng Oh, and Sang H. Son. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Transaction on Computers*, 44(12), December 1995. 8

[8] Liang Chen, K. Reddy, and Gagan Agrawal. GATES: A grid-based middleware for processing distributed data streams. In *Thirteenth IEEE International Symposium on High Performance Distributed Computing (HPDC-13)*, June 2004. 14

[9] M. Coli and P. Palazzari. Real time pipelined system design through simulated annealing. *Journal of Systems Architecture*, 42(6–7), December 1996. 7

[10] S.K. Dhall and C.L. Liu. On a real-time scheduling problem. *Journal of Operations Reserach*, 26(1):127–140, January–February 1978. 1, 4, 8

[11] Jr. E. G. Coffman. *Computer and Job-Shop Sheduling Theory*. John Wiley and Sons, New York, 1976. 1

[12] Michal Feldman, Kevin Lai, and Li Zhang. A price-anticipating resource allocation mechanism for distributed shared clusters. In *EC '05: Proceedings of the 6th ACM conference on Electronic commerce*, pages 127–136, New York, NY, USA, 2005. ACM Press. 12

[13] M. G. Harbour, M. H. Klein, and J. P. Lehoczky. Timing analysis for fixed-priority scheduling of hard real-time systems. *IEEE Transaction of Software Engineering*, 20(1):13–28, 1994. 1

[14] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, Michigan, 1975. 6

[15] Oscar H. Ibarra and Chul E. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *Journal of the ACM*, 24(2):280–289, 1977. 1

[16] Yuhui Jin and Rob Strom. Relational subscription middleware for Internet-scale. In *2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, 2003. 14

[17] Andrey Khorlin. Scheduling in distributed stream processing systems. Master's thesis, California Institute of Technology, 2006. 18

[18] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983. 7

[19] H. J. Siegel L. Wang, V. P. Roychowdhury, and A. A. Maciejewski. Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. *Journal of Parallel and Distributed Computing, Special Issue on Parallel Evolutionary Computing*, 47(1):8–22, 1997. 6

[20] J. Lehoczky. Fixed-priority scheduilng of task sets with arbitrary deadlines. In *11th Real-Time Systems Symposium*, December 1990. 8

[21] J.Y. T. Leung and J. Whitehead. On the complexity of fixed-priority scheduiling of periodic real-time tasks. *performance evaluation*, 2(2):237–250, February 1982. 10

[22] C.L. Liu and S. K. Khall. On a real-time scheduling problem. *Journal of Operations Research*, 26(1):127–140, February 1978. 9

[23] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard real time environment. *Journal of the ACM*, 20(1):46–61, January 1973. 1, 4, 8

[24] Andreu Mas-Colell, Michael D. Whinston, and Jerry R. Green. *Microeconomic Theory*. Oxford University Press, New York, 1995. 15, 23

[25] Jeonghoon Mo and Jean Walrand. Fair end-to-end window-based congestion control. *IEEE/ACM Transactions on Networking*, 8(5):556–567, October 2000. 31

[26] Scott Shenker. Fundamental design issues for the future internet. *IEEE Journal on Selected Areas in Communication*, 13(7), September 1995. 50

[27] Wei Kuan Shih, Jane W.-S. Liu, and C. L. Liu. Modified rate-monotonic algorithm for scheduling periodic jobs with deferred deadlines. *Software Engineering*, 19(12):1171–1179, 1993. 8

[28] A. Tang, J. Wang, and S. Low. Counter-intuitive behaviors in networks under end-to-end control. *IEEE/ACM Transactions on Networking*, 14(2), April 2006. 50

[29] Carl A. Waldspurger, Tad Hogg, Bernardo A. Huberman, Jeffrey O. Kephart, and W. Scott Stornetta. Spawn: A distributed computational economy. *IEEE Transactions on Software Engineering*, 18(2), February 1992. 10, 50

[30] Wikipedia. Grid computing. http://www.wikipedia.org/wiki/Grid_computing, May 2006. 50

[31] Rich Wolski, James S. Plank, John Brevik, and Todd Bryan. Analyzing market-based resource allocation strategies for the computational Grid. *The International Journal of High Performance Computing Applications*, 15(3):258–281, Fall 2001. 50