Dynamic Load-Balancing of StreamIt Cluster Computations

by

Eric Todd Fellheimer

B.S. Computer Science and Engineering, Physics
Massachusetts Institute of Technology, 2005

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2006

Author ....
Department of Electrical Engineering and Computer Science
May 26, 2006

Certified by ....
Una-May O'Reilly
Principal Research Scientist
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Dynamic Load-Balancing of StreamIt Cluster Computations

by

Eric Todd Fellheimer

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 2006, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis discusses the design and implementation of a dynamic load-balancing mechanism for computationally distributed programs running on a cluster written in the `StreamIt` programming language. `StreamIt` is useful for streaming data applications such as MPEG codecs. The structure of the language carries a lot of static information, such as data rates and computational hierarchy, and therefore lends itself well to parallelization. This work details a simulator for `StreamIt` cluster computations used to measure metrics such as throughput. Built on top of this simulation is an agent-based market used for load balancing the computation at `StreamIt` checkpoints to adapt to exogenously changing loads on the nodes of the cluster. The market models the structure of the computation as a supply chain. Our experiments study the throughput produced by the market compared to other policies, as well as qualitative features such as stability.

Thesis Supervisor: Una-May O'Reilly
Title: Principal Research Scientist

Contents

List of Figures

List of Programs and Files

Chapter 1

Introduction

The basis of modern day high-performance computing is parallel computation. In such computations, the work is split among various nodes, or processors, which can work simultaneously. Total efficiency of resource utilization (constantly using 100% of all the processors) is rarely achieved, however, due to dependencies among the work units on the various processing nodes.

Consider even the simple case of some computation involving two processing nodes, P_1 and P_2. Throughout the computation, P_1 computes data and P_2 processes that data further. If it takes P_1 exactly the same amount of time to produce data as it takes P_2 to process it, then this system will achieve total efficiency in the steady state. However, it is rare that two different processes will take the same amount of time, especially considering that they may be run on completely different processing units. Exogenous factors could also affect the computation. For instance, there may be other computations running on the same system, or there may be non-negligible communication time between the processing nodes.

Even if *load-balancing* (allocating the processing nodes to processors so that the processors are being used nearly the same amount) is used, exogenous factors (as mentioned above) could render static (compile-time) load-balancing ineffective. In order to achieve effective adaptability, the computation ought to employ *dynamic load-balancing*. That is, the computation must be able to reconfigure its processing nodes while it runs.

The project works with the stream programming language `StreamIt`. To facilitate easier experimentation with different load-balancing techniques, a simulator was created (section 4.1). On top of this simulator is an agent-based market for load-balancing the cluster computation while the cluster's nodes face exogenous load.

1.1 High-Level Problem Statement

The goal of this project is to devise a market-like, dynamic load-balancing system which adapts to changing load in a `StreamIt` (chapter 3) cluster computation. The main metric will be *throughput*: the total number of elements processed per unit time.

The market will be designed as a tradeoff between an "optimal" solution (which maintains high throughput, ignoring its large online overhead) and a static approach (which has no online overhead but is not adaptive). That is, the market ought to run efficiently so it doesn't take away too much resources from the main computation, while still providing useful adaptations to the system. The scope and specifics of these adaptations will be described in section 4.2.

1.2 Roadmap

This thesis starts with a chapter on related work (chapter 2). Most of the references cited in this chapter dealt with the load-balancing problem or with complex, distributed, and possibly agent-based systems. Chapter 3 discusses the `StreamIt` programming language. The next chapter, chapter 4, discusses the design and implementation of the `StreamIt` simulator, as well as the design and implementation of the load-balancing market. Experiments and results discussing stability and throughput improvements are discussed in chapter 5. Chapters 6 and 7 discuss further improvements and work to this line or research and general conclusions.

Chapter 2

Related Work

The relevant literature includes several examples of computational markets and agent-based systems. Their features are summarized in figure (2-1). The following sections summarize work related to this thesis and discuss important differences with our work.

	domain	funding	matching of buyers, sellers	commodity	adaptability	comments
Ferguson	CPU load balancing	Lump sum per job	Sealed bid or Dutch with local advertisements	time slices (fixed length)	none	Ignores queuing delays
Spawn	grid computing	constant rate per job	sealed-bid, second price	time slices	none	
Mirage	Sensornet Testbeds	Per user, sales tax, profit-sharing	Sophisticated combinatorial auction	resource combinations in time/space	none	
LeBaron	economic simulation	Initial endowment	price cleared explicitly	Risky stock	Agents adaptively select rules, rules are neural nets	
`StreamIt` Market	dynamic load-balancing stream computations	supply-chain distribution	Greedy market clearer	processing nodes	An agent's strategy depends on past results	See section 4.2

Figure 2-1: Summary of related work. "Domain" refers to the problem or research area motivating the work. "Funding" refers to how agents receive currency.

2.1 Ferguson

Ferguson uses microeconomic ideas to solve a load balancing problem[9]. He models the problem as a graph of processing units. Edges in the graph represent network connections. There is a bandwidth cost to send data between connected units. There is also an effective processor "speed" of each processing node. Each job starts at one of the processors and is not parallelized. However, it can migrate to other processors, at some cost, to support load balancing.

In the economy, jobs bid on processors given an estimate of the time they need. All jobs are given the same initial lump-sum endowment. They use this money to then bid on a processor, basing decisions both on frugality and quality of service. In Ferguson's work, jobs ignore queuing delays, time waiting for jobs to begin, instead trying to optimize *service time*, the time to finish the job once it begins.

Processors hold auctions for their use after advertising their recent price history to neighboring processors' *bulletin boards*. The system uses both sealed bid and Dutch auctions[1]. The auctions are decentralized, and processors may advertise in auctions at neighboring nodes.

This work is similar to the thesis in that it describes the resource set as a graph of processors and network connections, and that it focuses on load-balancing. One complication we deal with is that the filters in a `StreamIt` program are not in strict competition with one another. A "rational" filter would not starve its upstream neighbor of processing resources. If it did, it would never get data inputs and thus never be able to work.

2.2 Spawn

In `Spawn`[21, 11], the goal is to efficiently allocate resources for grid computation, perhaps across the Internet. Each job is given a steady rate of currency, its *funding rate*. Jobs can split themselves into different subjobs, but the total funding stays the

[1]In a Dutch auction, the price starts at some high value. Each round, the prices is decreased until some agent accepts and buys the good at the current price.

same. While the Spawn system does not constrain how funding gets divided among subjobs, all example code simply splits the funding evenly. Jobs and subjobs use their funding to bid on time slices at the various processing nodes. The relative funding rates in the system determine the relative priorities of different jobs.

CPU time slices are auctioned in a decentralized fashion at the various nodes, using a sealed-bid second price auctions[2]. Jobs are given a *right of first refusal* allowing them to continue paying the market prices for further time slices. This feature is not beneficial to the market efficiency, but rather required because technical limitations in the system do not allow processes to migrate.

If there are relatively few jobs in the system, the "market price"[3] will be lower and jobs, in general, will be able spawn and successfully bid on more processor time slices. Likewise, when there is vast competition, price will be higher and thus the same funding rate will not be able to buy as many concurrent time slices. Therefore, jobs will have a lesser tendency to split up and further divide their seemingly scarce funding.

The work is mainly applicable for highly paralellizable algorithms, such as a parallel Monte Carlo simulation. While StreamIt programs are paralellizable, it is not trivial to change the number of parallel processes used in a computation on the fly. Specifically, once each filter is running in parallel, the computation could not spawn additional processes even if the market allowed for it. A major difference between the Spawn system and our own is the funding policy. In Spawn, funding is distributed evenly among the work units in a given computation. In our system, agents adaptively determine how to distribute their funding.

[2]In these auctions, each agent privately submits his bid. The agent bidding the highest value receives the good, and pays the prices of the second highest bidder.

[3]Because there is an auction and not a commodities market, there is no explicit market price. However, competition will raise the bidding values, and thus the loosely-defined market price.

2.3 Mirage

`Mirage`[8] is a system for allocating resources in a SensorNet Testbed. Agents place *combinatorial* bids such as "I need 3 motes sometime next week" into a centralized auctioneer.

In `Mirage`, priority is represented in a user *share ratio.* Through *profit sharing*, the virtual currency returns to its equilibrium distribution (the currency is closed-loop and there is a fixed total amount). For instance, if a user "owns" a share ratio of 20%, then he or she gets 20% of each winning bid. Mirage also employs a "use it or lose it"[8, p. 5] in which a *sales tax* takes a percentage of a user's surplus over his or her equilibrium value. These two forces are complementary: Profit-sharing allows a low-priority user to accrue savings while higher-priority users deplete theirs, but sales taxing does not allow this advantage to continue indefinitely.

Here, bidding strategies are employed by the end user. Moreover, the bids occur in coarse time granularity. Resources are allocated on the order of hours, and bids are cleared on the order of minutes. The relatively long clearance time is a consequence of the relatively complex combinatorial auctions. This complexity would be unacceptable given the real-time nature of our system. The key difference between `Mirage` and this thesis is that `Mirage`'s goal is to create fairness in a distributed system while ours is to optimize the throughput of a parallel computation.

2.4 LeBaron

LeBaron's work on agent-based financial markets[15] is the only work cited here that truly adapts agents within the system. The various agents can purchase various amounts of a "risky security" within a commodities market at each time step. Their demand, which is a function of very recent market information, is represented by a neural net. This information, referred to as the *information set*, includes returns information and the price dividend ratio. The individual demands are summed, and the market is cleared at some price. Then the agents either benefit or suffer based on the state of the market and their most recent demand.

As the simulation progresses, agents are given the chance to change their rules, the neural nets which determine their demand function. Here, agents simulate what *would have happened* to their wealth had they been using some other rule. If the other rule appears to be performing better, the agent may swap out his current rule for this seemingly better one. These rules also evolve via a genetic algorithm. Thus, both the agents and the underlying rules evolve through time.

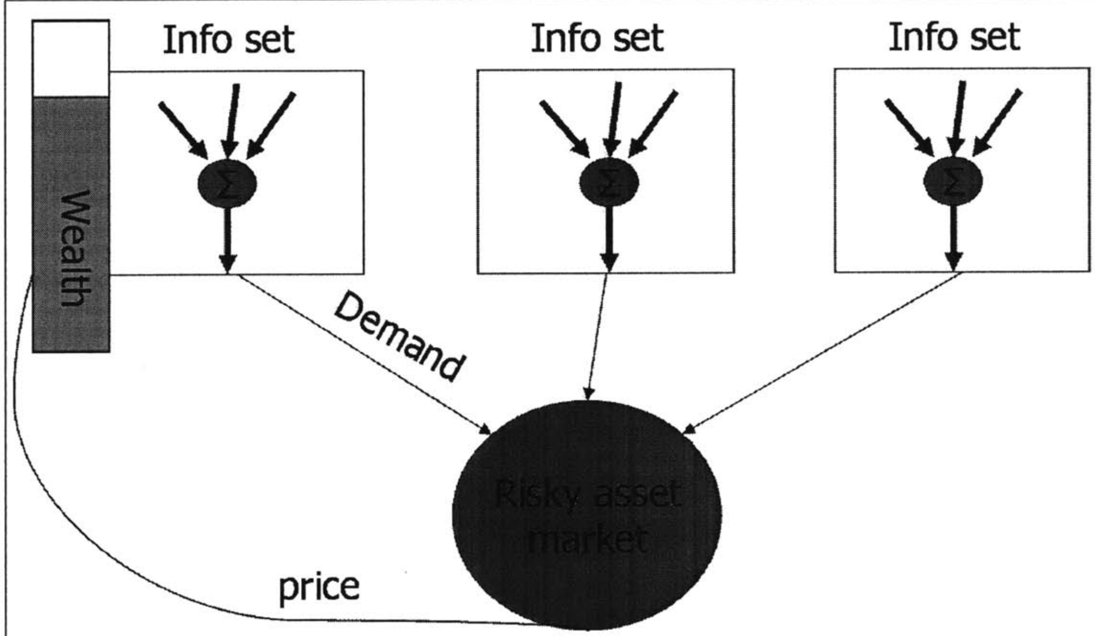


Figure 2-2: A diagram of the virtual marketplace in [15].

An important parameter in this historical simulation is the *memory length*. The memory length of an agent dictates how far back he simulates the market when comparing two rules. LeBaron goes on to discuss how different mixtures of shorter and longer memory length agents affect the dynamics of the market.

2.5 Hayek

In *Evolution of Cooperative Problem-Solving in an Artificial Economy*[4], the authors describe a general learning and problem-solving technique. Various agents work on solving a given problem and are assigned credit based on successful cooperation as well as individual contributions.

The `Hayek` artificial economy consists of computational agents who interact in a sequence of auctions. The agents simulate the impact on the problem being solved and return an estimate of value they would add. Agents bid based on their current wealth and this estimate. Wealthy agents will "reproduce" via mutation at certain times. Agents get a percentage of their offspring's income, and are taxed based on how many instructions they execute.

The `Hayek` system successfully solves difficult problem. Problems such as Blocks World have huge state spaces in which successful evaluator functions are nonlinear. Despite these difficulties, `Hayek` performs significantly better than competing techniques such as genetic algorithms. This work provides evidence that multi-agent, economically based systems can perform qualitatively differently (and better) than more traditional methods.

Chapter 3

StreamIt

The `StreamIt` programming language[20] enables a "compiler technology that enables a portable, high-level language to execute efficiently across a range of wire-exposed architectures."[10, p. 1]. In essence, the language is designed for programs to be written for and executed on newer computational architectures which employ vast parallelism and predictable communication. It is especially well suited for "streaming" applications such as signal processing.

3.1 The Language

A `StreamIt` program's most basic component is the *filter.* Filters describe single computation units with single input channels and single output channels. Filters contain *init*, *prework*, and *work* functions. Each is expressed in syntax similar to typical imperative languages such as C. Init sets up the filter, for instance by creating data table lookups. The prework function is called after the init function and before the steady state (i.e., work) is reached. The difference between prework and init is the prework may communicate with other filters.

The work function must specify *push*, *pop*, and *peek* values. *Push* determines the number of data elements the filter outputs after one execution of the work function. The filter consumes *pop* elements and reads *peek* elements every time it *fires.* Firing is used to mean a single, atomic execution of the filter's work function. The function can access the input elements from its input queue (or *buffer*) using the `peek(index)` and `pop()` operations. It writes to downstream queues using the `push(value)` operation. All filter queues are first-in first-out (FIFO) queues.

The structure of a `StreamIt` program is built by composing filters and compositions thereof hierarchically. We will refer to a single filter or some composition of filters from here on as a *stream*[10]. There are three constructs for composing streams. The *pipeline* construct simply sequentially attaches the sub-compositions. The output of the first is connected to the input of the second and so on.

A *splitjoin* creates a stream where the data go into a common stream, the *splitter*, diverge to various streams, and reconvene at the *joiner* stream. *Duplicate* splitters send a copy of each data element to all child streams. *Roundrobin* splitters, however, send a specific number of incoming data to the first child stream, the second child stream, and so on sequentially, until starting again at the first child stream. The *feedbackloop* mechanism allows loops to be created in the stream graph. Programmers may `enqueue` data values in feedback loops at the beginning of a computation. An example `StreamIt` program is shown in program 1, which produces the Fibonacci sequence.

3.2 Compile-Time Optimizations

The `StreamIt` compiler[10, 13] employs multiple techniques to improve runtime performance. The most relevant of which to this work is *partitioning*, which attempts to split the stream graph into a certain number of units which all have similar work requirements. We refer to this process later on as *static load-balancing* to differentiate from the dynamic load-balancing embodied in our computational market in section 4.2. The basic functionality is that "...the compiler estimates the number of instructions that are executed by each filter in one steady-state cycle of the entire program; then, computationally intensive filters can be split, and less demanding filters can be fused. Currently, a simple greedy algorithm is used to automatically select the targets..."[10, p. 5]. Partitioning is achieved through the use of both *fusion* (combining filters) and *fission* (splitting filters). Fission is a more difficult problem because it is similar to automatic parallelization in imperative languages. It is not implemented currently in the compiler[10, p. 7].

Another interesting class of optimizations are those which rely on the inherent memory model of the system[18]. Because the `StreamIt` language embodies rich static information such as data transfer rates and work estimates of filters, it can model and estimate cache behavior during a program execution. One of the cache optimizations is *scalar replacement*, which replaces buffer variables with scalars to improve register allocation. *Execution scaling*, on the other hand, repeats filter executions to improve instruction locality.

While our work ignores cache-related complications (see section 4.1.3), dynamic load-balancing does not preclude the use of cache optimizations like scalar replacement. Cache and other low-level optimizations can be used along with load-balancing for additional performance benefits.

```
void->void pipeline Fib {
    add feedbackloop {
        join roundrobin(0, 1);
        body PeekAdd();
        loop Identity<int>();
        split duplicate;
        enqueue 0;
        enqueue 1;
    };
    add IntPrinter();
}

int->int filter PeekAdd {
    work push 1 pop 1 peek 2 {
        push(peek(1) + pop());
    }
}

int->void filter IntPrinter {
    work pop 1 {
        println(pop());
    }
}
```

Program 1: A Fibonacci sequence generator in `StreamIt`. The feedback allows the `PeekAdd` filter to use its previous outputs as inputs. Code taken from the `StreamIt`repository.

Chapter 4

Experimental Setup

4.1 The StreamIt Simulator

A high level view of the StreamIt simulator's functionality is shown in figure 4-1. After compiling a stream program with the StreamIt compiler[1], an xml stream graph is generated. The stream graph and cluster graph are then input to the simulator.

The StreamIt Simulator is developed in the Java programming language. Java was chosen for its portability and ease of development. Also, the StreamIt compiler itself is implemented in Java, and it can produce Java code from StreamIt source code.

A major design goal during the development process was to treat the simulator not just as a case study in StreamIt optimizations, but also as a more general framework for studying multiple agents interacting. Therefore, we intended to create a highly modular system in which different components could be used without affecting the behavior of the rest of the system. For instance, we made the RuntimeHandler interface, which specifies the resource mapping of filters to processors. On top of this, we created several implementations of this interface to test various resource mapping policies. No matter which implementation we use, however, the rest of the system behaves correctly.

A list of command line switches for the simulator is shown in figure 4-2.

[1] We modified the compiler slightly to output stream graphs in our xml format.

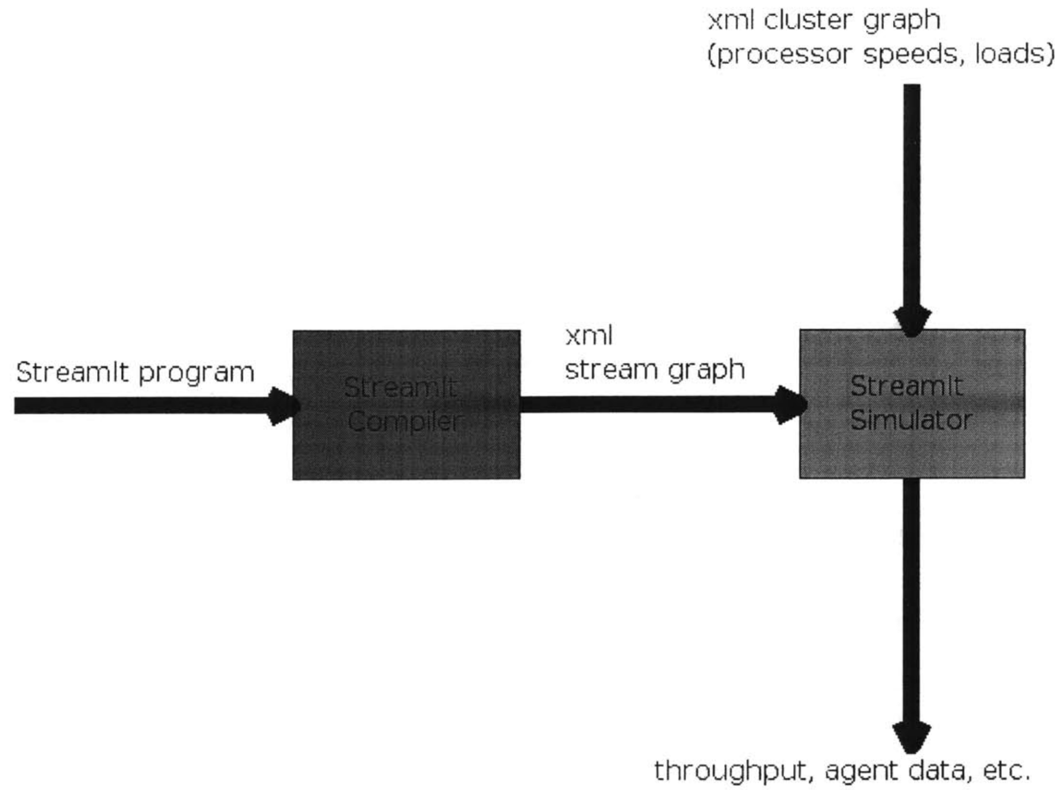


Figure 4-1: High-level functionality of the StreamIt simulator. A modified version of the compiler generates the stream graph xml file. That, along with the cluster graph xml file, are input to the simulator.

```
Options:
-cf, --cluster_file      Name of the cluster graph xml file
-cli, --cli              Output text to the standard output stream
-gui, --gui              Display the GUI
-n, --number_firings     Number of total filter firings to execute
-of, --out_file          Output simulation statistics to given file
-off, --offset-scale     Offset scale param for filter agents
-rep, --repetitions      How many times to repeat simulation
-rh, --runtime_handler Which runtime handler to use
-sf, --stream_file       Name of the StreamIt graph xml file
```

Figure 4-2: Important command line switches for the StreamIt Simulator.

The simulator is used to simulate a `StreamIt` cluster computation, a parallel computation in which the various filters are allocated to different processing nodes in the cluster. The simulator can track various metrics such as total throughput and time spent waiting for inputs for the individual filters.

4.1.1 The Model

??

The simulator uses a simplified model of the `StreamIt` language. Please refer to section 4.1.3 for a detailed review of the differences.

The *stream graph* encapsulates most of the information about the computation. It is a directed graph whose nodes are descriptions of filters. Edges are directed *downstream*, meaning in the direction of data flow. Thus, the *first vertex* has no parent vertices and the *final vertex* has no children vertices.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<graph edgedefault="directed">
        <node id="1" name="V1" />
        <node id="2" name="V2" pop="2" peek="3" work="7" />
        <node id="3" name="V3" peek="1" push="1" work="4" />
        <node id="4" name="V4" work="2" />
        <node id="5" name="V5" work="1" />


        <edge source="1" target="2" />
        <edge source="1" target="3" />
        <edge source="2" target="4" />
        <edge source="4" target="5" />
        <edge source="3" target="5" />

</graph>
```

Program 2: An example stream graph. Filter V_1 is the first vertex and filter V_5 is the final vertex.

When a stream graph such as the one in program 2 is parsed, it produces a `DirectedGraph`[12] whose vertices are of type `FilterVertex`. A `FilterVertex` contains dynamic information about the filter. Such information includes links to the filter's neighbors as well as a reference to its `InputBuffer`. Every filter has exactly one input buffer which it uses to store data elements as they arrive from *upstream* neighbors.

The `FilterVertex` also has a reference to the filter's static information stored in the `FilterDescriptor` class. The `FilterDescriptor` contains four data members. An atomic execution of a filter is referred to as a *firing*.

- **push**: The number of data elements output to downstream filters during each firing of the filter.
- **pop**: The number of data elements removed from the input buffer during firing of the filter.
- **peek**: The minimum number of data elements needed to fire.
- **work**: The estimated amount of work needed for a firing of this filter.

Another necessary input to the simulator is the *cluster graph*, which specifies the processors in the cluster and their properties. When a cluster file such as the one in program 3 is parsed, it generates a `DirectedGraph` whose vertices are instances of the `PCVertex` class. The `PCVertex` class contains dynamic information about the processor, such as which filters are currently executing on it. It also contains a link to the static processor information contained in instances of subtypes of the abstract `PCDescriptor` class. The `PCDescriptor` class contains the raw speed of the processor, which is its speed with no load.

Subclasses of `PCDescriptor` specify the *load model* of the processor. The load model returns the number of background processes running on the processors at a given time. Because the speed of the processor is estimated by taking its raw speed and dividing by the total number of processes running at a given time[2], the load model can be used to derive the speed of the processor at any given time in the simulation. The `StaticPC` subclass of `PCDescriptor` specifies a simple load model which always

[2]This should be a good estimate when the total number of processes remains somewhat constant, the operating system gives the filter approximately $\frac{1}{N}$ of the time slices where N is the total number of processes, and that the execution of the filter will take many time slices to finish.

```xml
<?xml version="1.0" encoding="iso-8859-1" ?>
<graph edgedefault="undirected">
        <node id="1" name="p1" speed="5" type="rand" />
        <node id="2" name="p2" speed="3" type="rand" />
        <node id="3" name="p3" speed="8" type="rand" />
</graph>
```

Program 3: An example cluster graph. Edges are not currently needed because network latency is currently not modeled (see section 4.1.3). The type parameter specifies the load model of the processor.

has 0 background processes. The `RandomProcPC` subclass of `PCDescriptor` specifies a randomized load model which flips between *loaded* and *unloaded* states. When loaded, the processor is likely to have many background processes although the actual number is randomized in both states.

Checkpoints and Runtime Handlers

Checkpoints occur periodically during the computation. At this point, the simulation flushes data elements in input buffers then runs the runtime handler. The `RuntimeHandler` interface specifies the method which selects a new *resource mapping*. The resource mapping determines which processor each filter is on. Every filter must be on exactly one of the processors. If the stream graph contains F filters and the cluster graph contains P processors, then the total number of possible resource mappings is P^F. The mapping is allowed to change at every checkpoint.

The `StaticLoadBalancer` implements the `RuntimeHandler` interface. It mimics the static load balancing done in the `StreamIt` compiler. The greedy algorithm it contains prioritizes filters based on their work estimate and maps them to the processor in order to equalize the total estimated time each processor needs to execute all its filters. Because of its static nature, `StaticLoadBalancer` returns the same mapping at every checkpoint.

The `DynamicLoadBalancer` is similar to the `StaticLoadBalancer`, except it uses the current, as opposed to raw, speed of the processor to select the best mapping.

It employs the same greedy approach as the `StaticLoadBalancer` does. It returns a different mapping at each checkpoint. The `DynamicLoadBalancer` represents a runtime handler which can choose near-optimal resource mappings with a large high overhead.

The `MarketRuntimeHandler` was created to be an adaptive runtime handler which chooses effective resource mappings with small overhead. It is discussed in detail in section 4.2.

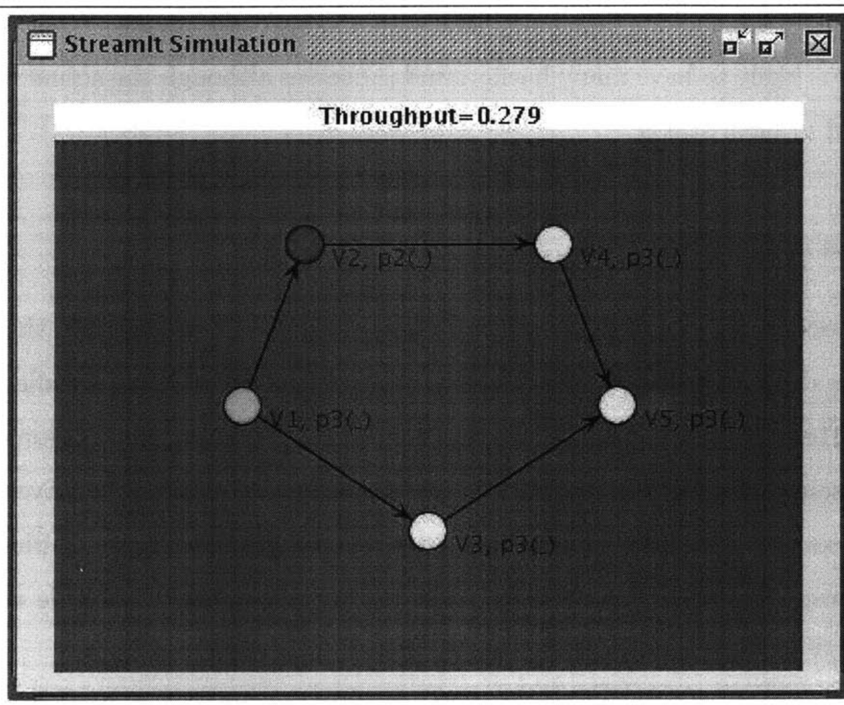


Figure 4-3: A graphical representation of the StreamIt simulator. The green node is the starting node (where the data originates). All other nodes are color coded: yellow means its input buffer is empty, red means its input buffer is full. One can see that the red node (V2) is a bottleneck because its buffer is full even though its immediate downstream neighbor (V4) has room in its input buffer. We also see that four of the filters have been allocated to processor $p3$.

4.1.2 Why Simulate

Experimenting with real `StreamIt` cluster computations could become quite cumbersome. Initiating the computations is still not as streamlined as it could be. Also, we do not have total control over the cluster during computation. We would have to create scripts to introduce load into the clusters when necessary. Moreover, we would like to be able to dictate the configuration of the cluster being used (which computers are on it and the properties of each of these) in order to experiment with many possible cluster arrangements.

Simulating these cluster computations provides much more power and flexibility. In simulation, we can specify the details of the cluster. We have utter control over how the processors become loaded. Moreover, we can run the entire simulation from a single computer with a single command.

4.1.3 Simulation Simplifications

The main problem this work tries to resolve is that of load balancing the `StreamIt` graph in an environment ridden with dynamic resource fluctuations. To this end, there were numerous components of a real `StreamIt` cluster computation which were not central to this problem.

- Instruction-Level Details

  The simulator works by estimating the firing time of filters in a "one-shot" manner based on processor speed, load, and computational work needed. A more comprehensive version might actually emulate the computation by going through each instruction in the computation. However, such a framework would be out of the scope of this work and completely change the architecture of the simulator. Nonetheless, using a more fine-grained approach would improve simulation accuracy. For instance, it would be easier to add a cache model to the simulation with instruction-level simulations.

- Network latency

  Currently, all network latency is assumed to be 0. Adding such effects would be rather simple though. Because the cluster is already specified as a graph, all that would be needed would be to associate some network latency model to each edge in the cluster graph, and add time samples of this load during filter firings in the simulation. We did not expect the addition of network latency to add to the richness of the market, and thus excluded this feature from the simulation.

- `StreamIt` Features Excluded in the Simulation

  The current implementation of `StreamIt` clustering does not provide support for feedback loops in the computation graph. Thus, this feature is not included in the simulation. Not having loops made the implementation of the simulator much simpler. For instance, in the `calcPrices` method of `FilterAgent`, recursive calls are executed on the filter's upstream neighbors. This algorithm surely terminates because there are no loops. If there were loops, the algorithm would have to add code to make sure it did not get stuck in infinite recursion.

  Additionally, the simulator does not maintain roundrobin splitters, only duplicate splitters. Roundrobin splitters allow a filter to inject into the same downstream filter multiple times before injecting into the next downstream filter. Adding such functionality would be quite easy, but would not add much richness to our simulation.

  The simulator does not take into account the prework functions of the various filters. We are mainly interested in the long-term throughput of a `StreamIt` computation, and the prework functions will most likely not contribute to the steady state computational efficiency.

  Additionally, variable rate filters do not exist in the simulation. All filters are modeled with constant push, pop, and peek values.

- Additional Overhead Not Included

  The most glaring shortcoming of the simulation is that it does not account for the overhead of the runtime handler. In other words, it assumes the runtime handler produces resource mappings instantly. Of course, the market runtime handler will incur some overhead. We assume that the checkpoint period is long and the overhead is minimal, and therefore the market's impact on overall throughput is small. One possibility to overcome this deficiency would be for the simulator to calculate work estimates based on Java bytecode. Because the `StreamIt` compiler can produce Java code from the filters, the simulator could compile all the filters and its own runtime handlers into bytecodes. Then, it would be able to estimate work times for the runtime handler and filters consistently.

A less severe shortcoming is excluding the overhead from the cluster runtime library. We assume such overhead to be small. We also do not account for migration times for checkpoints. That is, we leave out the time it takes for filters to move from one processor to another.

4.1.4 How It Works

The `StreamIt` simulator is not an instruction-level simulator. Instead of simulating a `StreamIt` filter's work line by line, it employs a *coarse-grained*, *data-driven* approach. Computation times are computed by using work estimates as well as information about the state of the processors. The system tracks data elements as they move through the graph. Each of these elements has an associated *timestamp*, which changes throughout the simulation.

The main loop of the simulation works by iteratively finding the next filter able to perform some action (which will either be *firing* or *injection*[3]). This process uses the `StreamItComputationDescriptor` class's `nextToRun` method. The `nextToRun` method simply iterates through all filters in the `StreamIt` graph, and runs the `timeToRun` method on each filter. It returns the filter which returned the smallest positive value from `nextToRun`. Negative values are returned when the individual filter does not have enough information to determine when it can next perform an action, which occurs as the result of two possible situations:

[3]An individual filter is either ready to fire, ready to inject, or neither. It can never be ready to inject and fire at the same time.

- The filter cannot inject into a downstream filter because the downstream filter's input buffer is full.
- The filter cannot fire because it does not have enough data elements (as specified by the peek attribute).

Pseudocode for the `timeToRun` method is shown in program 4.

Correctness Argument

Program 4 shows how filters determine when they can perform some event. We would like to be able to validate the correctness of the simulation. Our criterion is that of *temporal monotonicity*: filter *events* should be non-decreasing. Here, an event corresponds to either a filter firing or a filter injection. Thus, the simulation would be faulty if it first processes a firing at $t = 100$, and then a firing at $t = 50$.

At first, this correctness is not obvious. It seems possible that two filters would return -1 and 100 from their `timeToRun` methods. Thus, a firing would first occur at $t = 100$. But then, after this occurs, what if the other filter returns $t = 50$ on the next iteration. Then we would fire at $t = 50$ after we already fired at $t = 100$, a violation of our correctness condition. In the following, I will show that this and other violations cannot occur.

Lemma 4.1.1. *Each filter has temporal monotonicity.*

Proof. Each filter is in one of two states when its `timeToRun` method is called: waiting for injection or waiting for firing. Because a filter's event is only run after it returns a positive value from this method, we only need to look at the cases in which it returns a positive value. If the filter is waiting to fire and returns a positive value, then this value is at least as large as the last time the filter injected(see `nextPushTime`). Thus, monotonicity is maintained for firings.

If the filter is waiting to inject and the method returns a positive value, then we see that that the value it returns is at least as large as the previous firing completion time (not shown in the pseudocode), and the previous injection time (because `nextPushTime` is non-decreasing). Thus, monotonicity is maintained for injections.

```
//nextPushTime previously set to the time this filter last
   ...finished firing
if (filter is still pushing output downstream)
{
    if(canPushNextOutput())
    {
        if(waitForSpace)
        {
            //wait for the downstream filter to finish firing
            nextPushTime = Max(nextPushTime,
               ...downStreamDoneCompTime());
            waitForRoom = false;
        }

        return nextPushTime;
    }
    else //can't push next output, don't know when we will be
       ...able to, return -1
    {
        waitForSpace = true;
        return -1;
    }
}
else
{
    //check if we have enough elements to fire
    if (buffer.size() >= filterDesc.getPeek())
    {
        //if so, return the appropriate time based on data
           ...element times and previous completion of injections
        return Max(buffer.getTimeAt(filterDesc.getPeek() - 1),
           ...nextPushTime);
    }
    else //don't have enough elements, and not sure when they
       ...will come in, so return -1
    {
        return -1;      // can't fire for indefinite time
                        // not enough data elements!
    }
}
```

Program 4: Pseudocode for method `timeToRun` in class `FilterVertex`. The method may return a negative value under two circumstances. The first occurs when the filter must inject into an input buffer which is full. The second is occurs when the filter cannot fire because it does not have enough elements in its own input buffer.

Thus, Monotonicity is maintained in both cases. □

Now we must only show that monotonicity is maintained between any two pairs of *different* filters. Let the sequence of n simulation events occur at times $t = e_1$, $t = e_2$, ..., $t = e_n$.

Theorem 4.1.2 (Simulation Correctness). *Event e_x occurs at the same time or before e_y in the simulation if $y = x + 1$.*

Proof. This proof is by contradiction. Suppose $y = x + 1$ and that $e_x > e_y$, which must exist if simulation correctness is disobeyed.

If there are no negative values returned by `nextPushTime`, we see that there can be no out of order simulation events. This is due to lemma 4.1.1. Thus, the out of order events must occur after some negative value is returned. There are two possible cases:

- **Not enough data elements to fire**

  In this case, we have a firing at $t = e_y$ (let the filter that fires at this point F_y) and $e_y < e_x$, where $y = x + 1$. In the iteration in which event e_x is run (let the associated filter be F_x, we know that filter F_y returns a negative value from `timeToRun` due to lemma 4.1.1. If F_x injects enough elements for F_y to fire at $t = e_x$, then we know $e_x \leq e_y$ which contradicts $e_x > e_y$. If F_x does not do this, then there must be some event in between e_x and e_y which does inject into e_y, however this contradicts $y = x + 1$. Thus, there is some contradiction.

- **Cannot inject into filled downstream input buffer**

  In this case, we have an injection at $t = e_y$ (let the filter that injects at this point F_y) and $e_y < e_x$, where $y = x + 1$. In the iteration in which event e_x is run (let the associated filter be F_x, we know that filter F_y returns a negative value from `timeToRun` due to lemma 4.1.1. If F_x fires at $t = e_x$, then we know $e_x \leq e_y$ which contradicts $e_x > e_y$. If F_x does not do this, then there must be some event in between e_x and e_y which does remove from the input buffer of F_x, however this contradicts $y = x + 1$. Thus, there is some contradiction.

Both cases lead to a contradiction, so the initial assumption must be false. □

4.1.5 Testing and Debugging

Testing and validation occurred in two major steps. First, simple test cases were used as basic "sanity checks." A simple stream graph pipeline was created, as well as a simple cluster graph. The processors were given no exogenous load, and static load balancing was used. Once we output the resource mapping, we were able to calculate the theoretical throughput of the system. When the system was run, its throughput did converge to the theoretical steady-state throughput[4].

The second step was the liberal use of Java assertions. For example, in the `InputBuffer` class, there is an assertion making sure that the elements in the buffer are in correct temporal sorted order. The `nextToRun` method of the `FilterVertex` class makes the most important assertion: it asserts that whenever a filter is run, the corresponding time is greater than or equal to the previous run of a filter. This assertion therefore provides empirical evidence for the correctness of the program, fortifying the proof in section 4.1.4.

The graphical interface was also somewhat helpful during the testing and development process. Viewing the stream graph on the screen allowed us to quickly verify that the corresponding file had been parsed correctly.

4.2 The Market

The `MarketRuntimeHandler` class is an implementation of the `RuntimeHandler` interface which is meant to be an adaptive, low-overhead resource mapping mechanism. Currently, the implementation is a very simplistic subset of true computational market complexities. Nonetheless, we feel it provides insights into how a multi-agent system might help in resource allocation problems.

The market works as follows. The final vertex is given a revenue allotment of

[4]The convergence is due to the initial cost of having to fill up the input buffers prior to steady state execution and the initial latency.

1 monetary unit. The final vertex then divides this revenue among its upstream neighbors and its *resource budget*. For instance, if the final vertex has upstream neighbors B and D, it could allot .4 to B, .5 to D, and .1 to its resource budget. This continues until all filters have divided their revenues. Thus, B divides its .4 among its upstream neighbors and a resource budget. This process can be seen in figure 4-4. Strategies on how exactly the agents split their revenues are determined by the `FilterAgent` class and its subclasses.

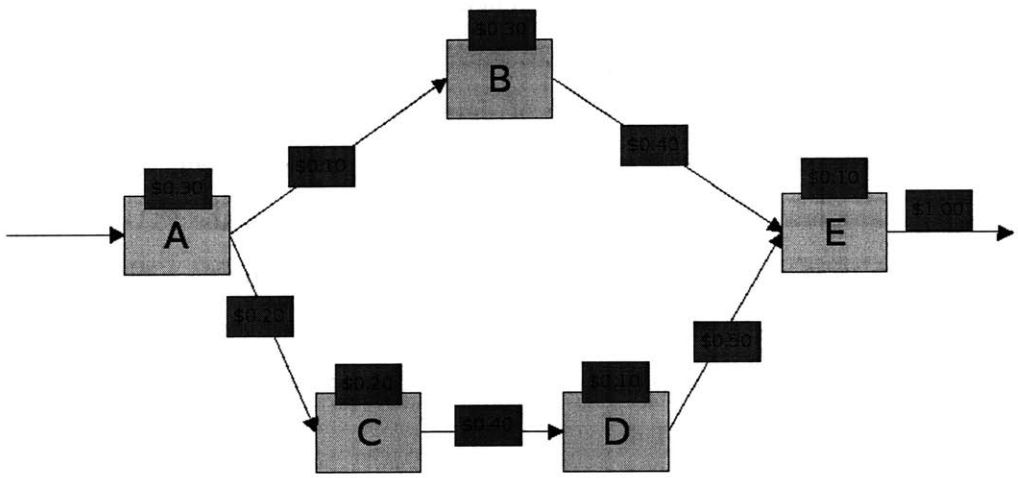


Figure 4-4: Monetary distributions. Lettered boxes are filters. Green boxes represent revenue distributions. Green boxes just above the filters are the associated resource budget. The total sum of all the resource budgets will always add to the revenue of the final vertex, which is arbitrarily set to 1 monetary unit. Revenue allocation occurs in sequence from the final vertex upstream to the start vertex.

Once all filters have calculated their resource budgets, these are input into a centralized market clearing mechanism. The market clearance is accomplished via a greedy algorithm (see the `getBestMap()` method of the `MarketRuntimeHandler` class) which uses the filters' resource budgets as indicators of priority. Thus, a filter which allots twice as much to its resource budget as another would expect to be run on a processor twice as fast as the other filter (or on a less loaded processor). Refer to program 5 for code relating to the market clearance mechanism.

```java
/**
 * A heuristic evaluation of a mapping based on market balancing.
 * @param map The mapping of filter to resource (initial greedy selections)
 * @return a score of how good the mapping is (the lower, the better)
 */
private double evaluateMapping(Map<FilterVertex, PCVertex> map, double t)
{
    final Collection<FilterAgent> agentCol = new LinkedHashSet<FilterAgent>();
    for(FilterVertex v : map.keySet())
    {
            agentCol.add(filterAgents.get(v));
    }

    final Map<FilterAgent, Double> agentSpeedInMap = new LinkedHashMap<
        ...FilterAgent, Double>();
    final Map<FilterAgent, Double> agentSpendingMap = new LinkedHashMap<
        ...FilterAgent, Double>();

    //populate the agentSpeedInMap and agentSpendingMap mappings
    for(FilterAgent a : agentCol)
    {
            //System.out.println(a + "...." + map.get(a.getFilterVertex()));
            final double mySpeedInThisMap = getSpeedInMap(map, a, t);

            agentSpeedInMap.put(a, mySpeedInThisMap);
            agentSpendingMap.put(a, a.getResourceBudget());
    }

    final double agentSpeedMin = Collections.min(agentSpeedInMap.values());
    final double agentSpendingMin = Collections.min(agentSpendingMap.values());


    double score = 0;

    //calculate error sum of ratios
    for(FilterAgent a : agentCol)
    {
            final double r1 = agentSpeedInMap.get(a) / agentSpeedMin;
            final double r2 = agentSpendingMap.get(a) / agentSpendingMin;

            final double diff = r1 - r2;
            score += diff*diff;
    }


    //calculate total processing speed of used processors
    final double totalSpeed = Util.Sum(agentSpeedInMap.values());

    //divide by total speed means we'll use more of the faster procs
    final double result = score / (totalSpeed * totalSpeed * totalSpeed) ;

    return result;
}
```

Program 5: The `evaluateMapping()` method of the `MarketRuntimeHandler` class is used in the greedy market clearing algorithm to determine the best greedy mapping choice at each stage.

An astute reader might note a potential problem with this scheme. If the final vertex allots almost all of its revenue to its own budget, then it will be of much higher priority than any other filter, hindering global efficiency. Luckily, this is not a real problem: even though each filter attempts to greedily maximize its own throughput, it "knows" that it will surely perform poorly if its upstream neighbors rarely provide input or if its downstream neighbors cannot handle its output (when their input buffers fill).

The revenue distribution algorithm resides in the `distributeRevenue` method of the `AdaptiveFilterAgent` class. The main metric agents use is the fraction of time in the previous checkpoint spent waiting for data. If it is high, data is not coming in fast enough, so the agent will lower its resource allocation, allowing upstream filters to gain priority. It the fraction is too low, then the filter is likely to be not keeping up with the influx of data elements. Thus, it increases its resource budget. More extreme values of the data wait fraction will elicit greater budget changes, but only to a certain extent. Agents are restricted in how much they can change their allocation at each checkpoint in order to facilitate stability. Pseudocode for the revenue distribution is shown in program 6 along with a description of the relevant parameters, `OFFSET` and `OFFSET_SCALE`.

While we have presented two specific policies for market clearance and revenue distribution, there remain many other policies waiting to be explored. We hope to explore other policies as well as the parameterization space of the current policies. Please refer to chapter 6 for further discussion on possible avenues of future research.

4.2.1 Market Runtime Analysis

Let the stream graph have F filter nodes and the cluster graph have P processor nodes. Then the runtime of the market clearance mechanism, as described above is approximately $\mathcal{O}(PF^2)$. This behavior is a consequence of the greedy algorithm. At a high level, the algorithm has F iterations, each of which runs in order $\mathcal{O}(PF)$, hence the total runtime behavior of $\mathcal{O}(PF^2)$. This runtime seems reasonable, even for relatively larger stream and cluster graphs. Nonetheless, we predict significant

```
oldResourceBudgetFrac = 0.5;

Function RevenueDistribution(revenue) returns resourceBudget
        dataWaitFracOfCheckpointTime = dataWaitDuration /
           ...checkpointDuration;
        error = dataWaitFracOfCheckpointTime - OFFSET;
        correction = error * OFFSET_SCALE;
        resourceBudgetFrac = oldResourceBudgetFrac - normalized(
           ...correction);
        oldResourceBudgetFrac = resourceBudgetFrac;
        return resourceBudgetFrac * revenue;
```

Program 6: Revenue distribution pseudocode. The code works by modifying the fraction of its revenue which it allots to upstream inputs each checkpoint. The main metric used here is the `dataWaitFracOfCheckpointTime`, the fraction of time in the previous checkpoint period spent waiting for input data. This method is parameterized by two values, the `OFFSET` and the `OFFSET_SCALE`. The `OFFSET` is the cutoff value for the `dataWaitFracOfCheckpointTime` metric. If `dataWaitFracOfCheckpointTime` is above `OFFSET`, the filter has spent too much time waiting. The `OFFSET_SCALE` determines how much the new fraction of input money to upstream filters can be modified at each checkpoint.

room for improvements in this algorithm by using results from past runs or other heuristics.

Chapter 5

Experiments and Results

It would not be feasible to search the entire parameter space of the StreamIt simulator during experimentation. The layout of the stream graph along with the features of the filters, the speed and load model of the processors, the runtime handler, as well as several runtime handler related parameters can all be specified as inputs to the simulator. Therefore, we concentrated on two major notions during the experimentation process. First, the market ought to produce better throughput than the StaticLoadBalancer, but not as good throughput as the DynamicLoadBalancer. Secondly, we were interested in the complex and adaptive behavior of the economy of agents. Is the market stable? How quickly does it react to catastrophic changes in load? We pursue these questions in the following sections.

5.1 Testing Framework

The testing framework is a set of python scripts. These scripts run the StreamIt Simulator on various inputs and accrue the results. In most cases, the python script will generate some data which is read by a simple gnuplot script which plots the results. Moreover, the simulator itself produces several plots each time it runs. These plots contain single-execution behavior, such as the resource budget each filter sets at each checkpoint in the MarketRuntimeHandler.

5.1.1 Experimental Parameters

The output and behavior of each simulator run is determined by the set of parameters passed in. The relevant parameters include the stream graph, the cluster graph, the runtime handler, revenue distribution parameters (only if using the market).

All experiments used the stream graph shown in program 7. The experiments also use the same cluster graph as seen in program 3, except with varying load models (refer to section ?? for a discussion on load models). None of the experiments change the market clearance mechanism.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<graph edgedefault="directed">
        <node id="1" name="V1" FIRST_VERTEX="" />
        <node id="2" name="V2" work="7" />
        <node id="3" name="V3" work="4" />
        <node id="4" name="V4" work="2" />
        <node id="5" name="V5" work="1" LAST_VERTEX="" />


        <edge source="1" target="2" />
        <edge source="1" target="3" />
        <edge source="2" target="4" />
        <edge source="4" target="5" />
        <edge source="3" target="5" />

</graph>
```

Program 7: The simple stream graph used in all experiments.

5.2 No Exogenous Load

In this experiment, our goal was to use a simple cluster graph with three processors of varying speeds with no exogenous load. Such a setup allows us to study market volatility (i.e., resource budgets, resource mappings, throughput) in a static environment. We also wanted to see how the throughput of the various runtime handlers would compare in a static environment. We know the policy we have given each agent (see program 6) is adaptive. When filters assess their local state, they will respond with different resource budgets (interpreted as prices by the market clearer). Our goal is to assess the macroscopic stability or volatility of these dynamics from checkpoint to checkpoint.

This experiment used static load for all three processors. We ran the simulator with the static and dynamic load balancers. We also performed a sensitivity analysis on the OFFSET and OFFSET_SCALE parameters of the MarketRuntimeHandler. Throughput results are shown in table 5-11.

runtime handler	throughput
static	1.1969
dynamic	1.1969
market(OFFSET=.2, OFFSET_SCALE=.25)	1.0978

Figure 5-1: Results from the no exogenous load experiment.

The market, in terms of throughput, performs 91.7% as well as the other two runtime handlers, for the two parameters which maximized its throughput. This seems reasonable because the market has nothing to adapt to. Also, there will be some startup, stabilization time at the start of the market before steady-state behavior. Additionally, that the static and dynamic runtime handlers performed exactly the same makes sense: when there is no exogenous load, these two policies perform almost exactly the same.

A sensitivity analysis of the market parameters is shown in figure 5-2. From left to right (varying OFFSET), there is a striking sensitivity. The best throughput values occur close to OFFSET=.2. Recall that the OFFSET parameter determines a level of satisfaction with a given input data waiting time. Therefore, one may view decreasing values of OFFSET as increasing levels of "greed." If OFFSET=.5, then the agents are content with waiting for data half the time. Performance will be poor because agents will not react correctly to high data wait times. If OFFSET=0, then the agents are only satisfied with no data waiting time. They will all spend all of their revenue on resource budget, and performance will degrade. Thus, in this case, it appears that

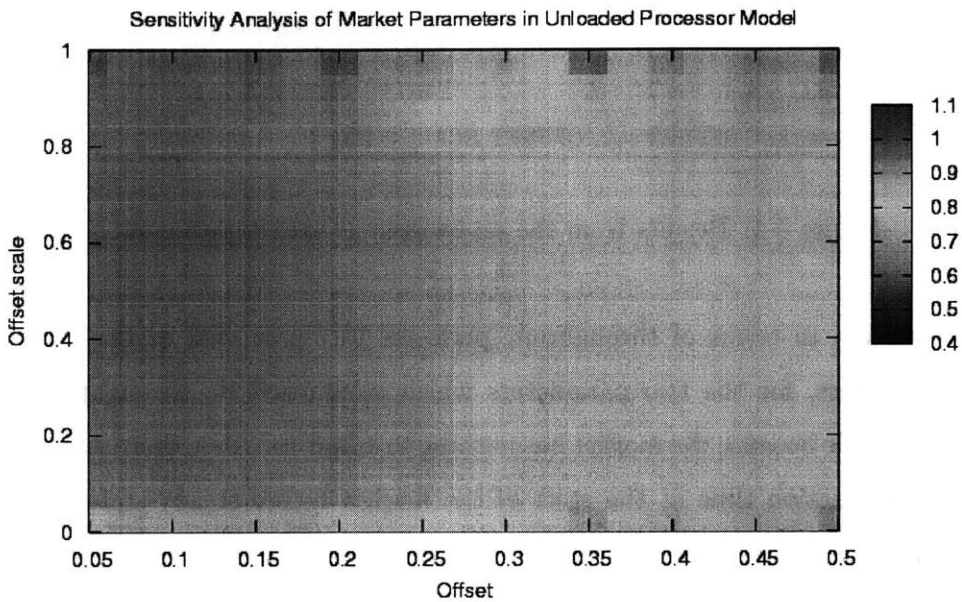


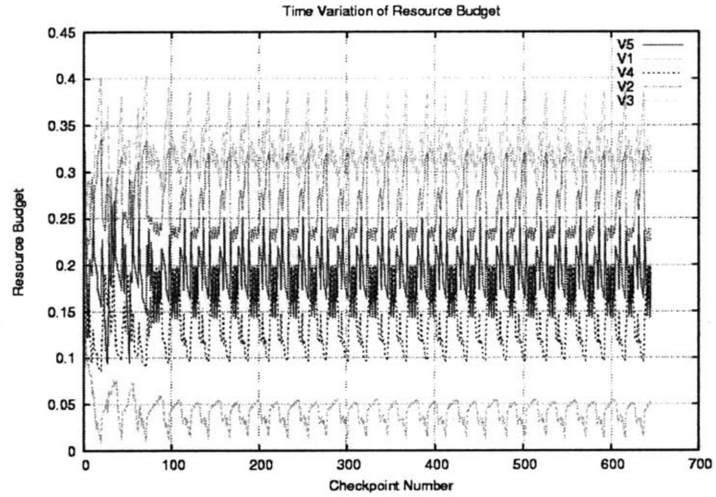
Figure 5-2: OFFSET and OFFSET_SCALE sensitivity analysis for the market with no exogenous load. The variables were modified in increments of .05. The color coding shows the various levels of throughput.

OFFSET=.2 provides the right balance between greed and apathy.

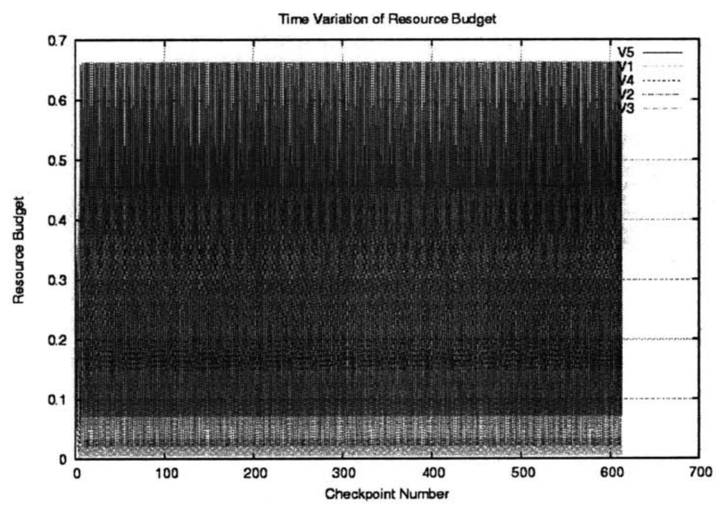
From bottom to top (varying OFFSET_SCALE), one can make out similar patterns. If OFFSET_SCALE=0, agents will never change their resource budgets, and thus the market will not react to anything. If OFFSET_SCALE=1, the agents will be able to change their resource budgets significantly at each checkpoint. This will hinder performance due to unwieldy market volatility. Specifically, if the resource budgets change too quickly, using the previous checkpoint period to speculate on the future will provide poor results.

Figure 5-3 compares the resource budget values for the filters versus the checkpoint number for two different combinations of market parameters. Figure 5-3(a) shows the plot for the parameter combination which maximized throughput, while 5-3(b) shows the plot for the parameter combination which produced very poor throughput. It is interesting to note the regular, cyclic behavior in both of these plots. The second plot displays immense volatility, as it comes from a high value of OFFSET_SCALE. Because the resource budget is a measure of filter priority, it makes sense that the right plot would show immense volatility in the resource mappings generated by the market clearance and that its performance is poor.

Figure 5-4 shows the same two market parameter combinations as above, but graphs the time evolution of throughput in both cases. In 5-4(b), the volatility of the resource mappings is manifest in the throughput volatility, and net throughput is low. However, in 5-4(a), the market is able to maintain longer periods of higher throughput. Thus, the net throughput is better than in the other case. Nonetheless, there is still significant volatility in throughput considering the lack of exogenous load. This phenomenon most likely relates to the brittle nature of the market clearing mechanism, which makes no guarantees that a small change in resource budgets will produce a "small" change to the resource mappings.

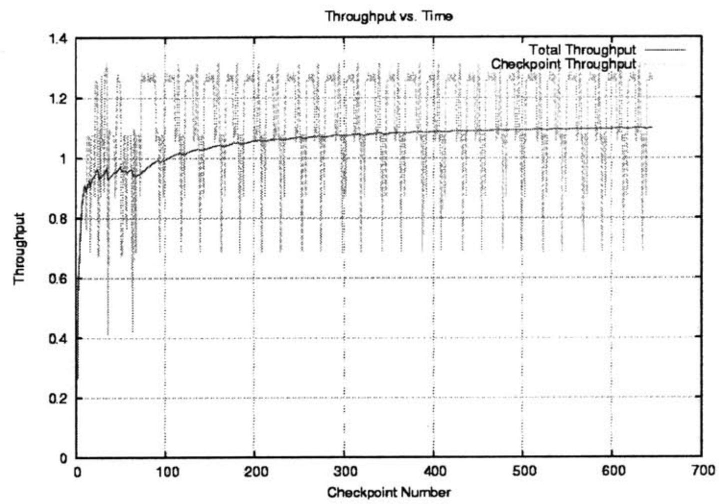


(a) OFFSET=.2, OFFSET_SCALE=.25

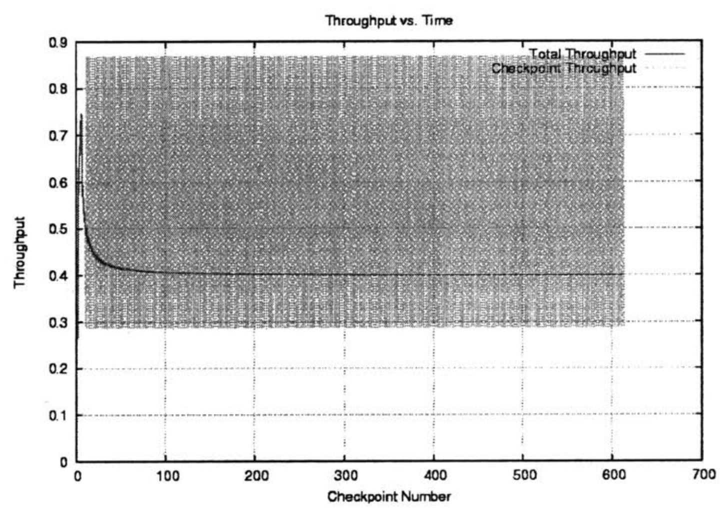


(b) OFFSET=.4, OFFSET_SCALE=.95

Figure 5-3: Resource Budget versus checkpoint number for different market parameter combinations



(a) OFFSET=.2, OFFSET_SCALE=.25



(b) OFFSET=.4, OFFSET_SCALE=.95

Figure 5-4: Throughput versus checkpoint number for different market parameter combinations

5.3 The Epsilon Parameter

After the experiments in 5.2, we wanted to see if there was some way to improve the stability, and thus the throughput. To this end, we added an `epsilon`(ϵ) parameter to the revenue distribution policy of the agents. This parameter provides a "cushion" around the OFFSET. Within this cushion, the agent will make no change to her resource budget percentage of revenue. An updated view of the resource distribution pseudocode is shown in program 8.

```
oldResourceBudgetFrac = 0.5;

Function RevenueDistribution(revenue) returns resourceBudget
        dataWaitFracOfCheckpointTime = dataWaitDuration /
           ...checkpointDuration;
        error = dataWaitFracOfCheckpointTime - OFFSET;

        if ( abs(error) < EPSILON)
                resourceBudgetFrac = oldResourceBudgetFrac
        else
                correction = error * OFFSET_SCALE;
                resourceBudgetFrac = oldResourceBudgetFrac -
                   ...normalized(correction);
        endif

        oldResourceBudgetFrac = resourceBudgetFrac;
        return resourceBudgetFrac * revenue;
```

Program 8: Revenue distribution pseudocode with ϵ.

We used the same setup as that of the last experiment, and fixed OFFSET=.2, OFFSET_SCALE=.25. Figure 5-5 shows a plot of throughput versus the ϵ parameter. In all previous experiments, effectively $\epsilon = 0$. Thus, we see that for small values of the parameter, for $.03 \leq \epsilon \leq .04$, the throughput actually improves (and even does better than the static and dynamic load balancers). For higher values of this parameter, performance drops precipitously. When the parameter is small, the cushion provides added stability without dramatically affecting the agent's preferences. However, when ϵ gets larger, agents will effectively become less particular about data wait times, and the market will be less adaptive.

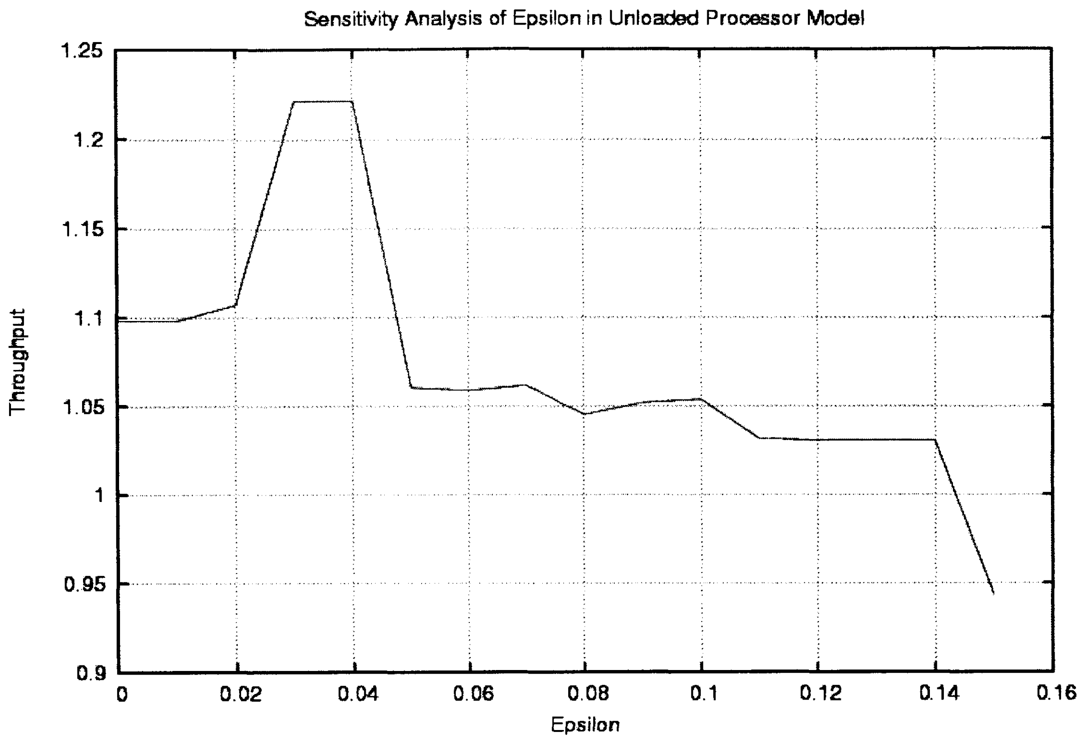


Figure 5-5: ϵ sensitivity analysis for the market with no exogenous load. The variable was modified in increments of .01.

In order to get a better sense of the nature of varying ϵ, we plotted execution data for the runs with three different ϵ values. The lowest values ($\epsilon \leq 0.02$) produced throughput similar to when the parameter did not exist. The middle range ($.03 \leq \epsilon \leq 0.04$) produced the best results, and the higher values ($\epsilon \geq 0.05$) produced poor throughput results. In figure 5-6 we see plots of the resource budgets for the three different values. Each displays qualitatively different behavior. Figure 5-6(a) looks like the typical, cyclical behavior we have already studied in figure 5-3(a). In figure 5-6(b), however, the amplitude of budgetary fluctuations has dropped dramatically, and the market ought to be much more stable. Figure 5-6(c) shows a market which is too lax. It has reached a steady state in which all agents are content (even though its corresponding throughput is not impressive). Because ϵ is too high in this case, the agents are not picky enough over their own performance, and thus global performance degrades.

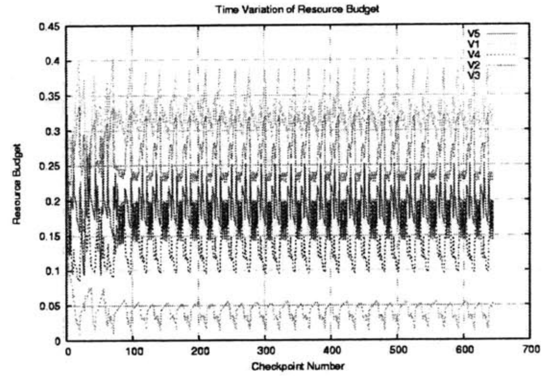
Figure 5-7 shows the effects of the ϵ parameter on the throughput at each checkpoint. We are now able to truly see the positive effects the ϵ parameter can have on the system in figure 5-7(b). Because of the added stability, the market clearance does not make dramatically different resource mappings. Therefore, when the market adapts and reaches a good mapping, it is able to retain this high throughput and perform quite well. Figure 5-7(c) shows the embodiment of too little adaptability. The market fails to improve upon a poorly performing resource mapping.

5.4 Singular Load Change

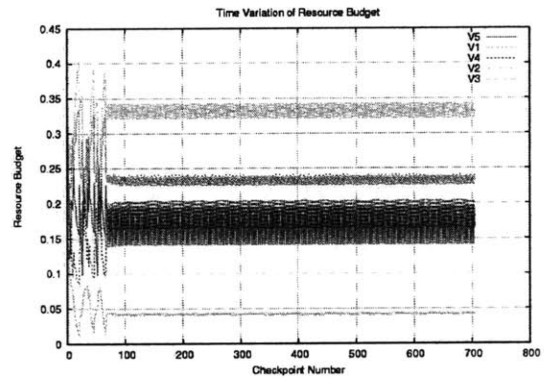
In this experiment, there is a simple load change in one of the processors. After checkpoint 400, the processor with raw speed 8 has 5 exogenous processes. This could model some abrupt change in the cluster environment, such as when another user starts a large, time-consuming process on one of the machines. The other two processors remain with the same static load. We run the three runtime handlers, using the market parameters which produced the greatest throughput from the first experiments (OFFSET=.2, OFFSET_SCALE=.25). The static load balancer ought to perform poorly in this case because it optimizes the resource mapping for a cluster which will change dramatically. We expected the dynamic load balancer and market to perform well, as they ought to be able to adapt to the changing environment.

Table 5-8 summarizes the results from this experiment. The dynamic load balancer performed much better than the static load balancer, as expected. Surprisingly, though, was the weak throughput measure while using the market. It's throughput was actually *lower* than in the static case.

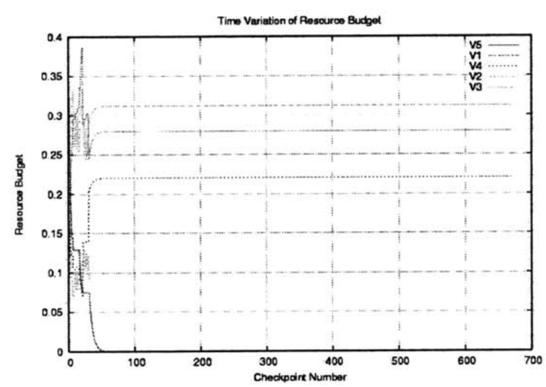
Figure 5-9 shows plots of the throughput measurements for the three runtime handlers. Surprisingly, we see little market fluctuations in figure 5-9(a) after the single processor load changes abruptly at checkpoint 400. We normally see a lot of fluctuation in throughput as the market constantly adjusts itself.



(a) $\epsilon = .01$

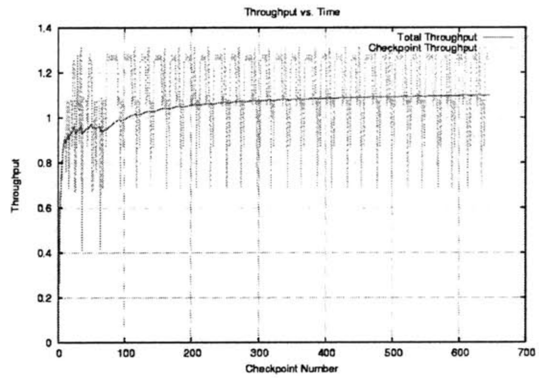


(b) $\epsilon = .03$

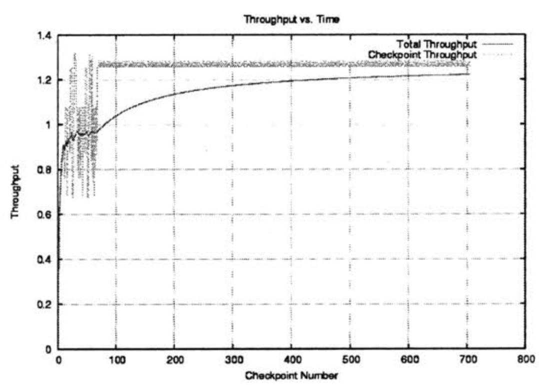


(c) $\epsilon = .12$

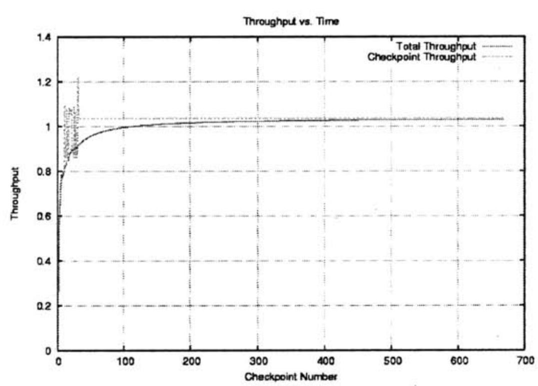
Figure 5-6: Resource budget versus checkpoint number for different ϵ values



(a) $\epsilon = .01$



(b) $\epsilon = .03$



(c) $\epsilon = .12$

Figure 5-7: Throughput versus checkpoint number for different ϵ values

runtime handler	throughput
static	0.439410
dynamic	0.728940
market(OFFSET=.2, OFFSET_SCALE=.25)	0.326990

Figure 5-8: Results from the singular load change experiment

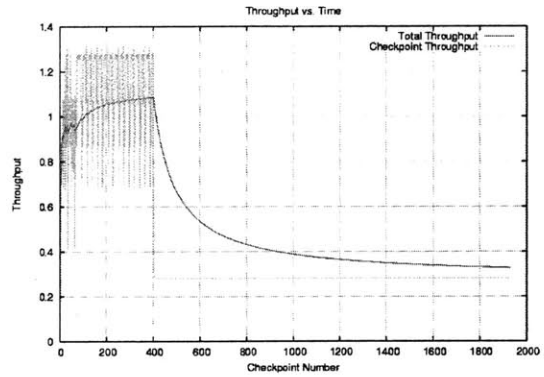
To get a better feel for what was going on inside the market, we plotted the resource budget percentages in figure 5-10. Immediately, one can see the degenerate nature of this run. All budget allocations quickly converge to either 0 or 1 after checkpoint 400. What could cause such a situation? We see that the final vertex V_5 passes on all of its revenue to its upstream neighbors. Thus, it has a resource budget of 0. Despite it giving itself low priority, it spends very little time waiting for inputs. Filter V_2 and V_3, however, consume the entire revenue pool as each has a budget of $\frac{1}{2}$. They are the top priority filters, yet they spend much of their time waiting for inputs.

We tracked down the issue to the greedy market clearance mechanism. It turned out that the greedy version had some poor properties. Because the algorithm does not explore the entire resource mapping space, of course it will not always return the "optimal" solution[1]. More importantly, the algorithm does not guarantee *fairness*. Here, fairness means that if filter A's resource budget is greater than filter B's, then filter A runs at least as fast as filter B[2].

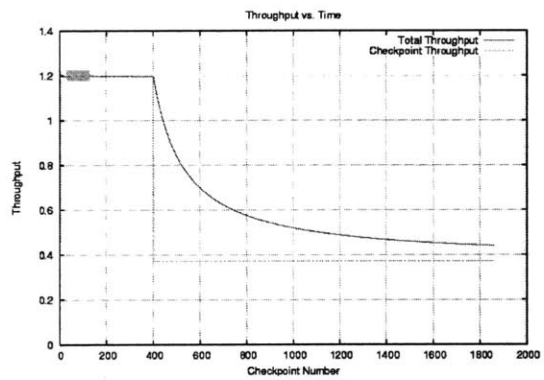
We hypothesize that the fairness property provides for the feedback which maintains stability in the market. That is, when there is fairness, resource budgets of the agents converge to certain values. Agents who have low data wait times (below OFFSET) increase their resource budget, and agents who have high data wait times decrease their budgets. After this budget change happens for several checkpoints

[1]Here, the optimal solution is the one which minimizes the difference between the ratio of the filters' speeds in the mapping and the ratios of the filters' resource budgets. There is also a small factor which tries to make sure the allocation takes advantage of fast processing resources.

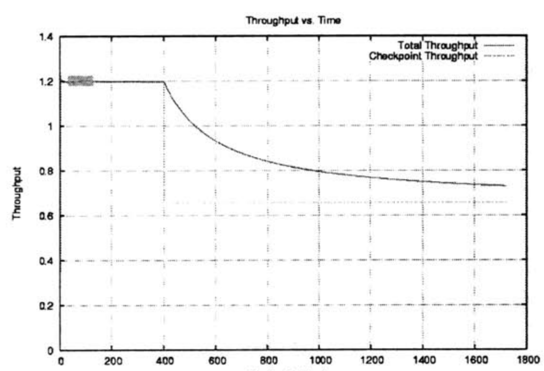
[2]We must be careful about what "speed" means. Here, it means that the raw speed of the processor, divided by sum of the total number of filters on the processor and the exogenous processes. Thus, speed = $\frac{R}{F+E}$, where R is the raw speed, F is the number of filters on the processor, and E is the number of exogenous processes.



(a) market



(b) static



(c) dynamic

Figure 5-9: Throughput plots for the different runtime handlers responding to singular load change

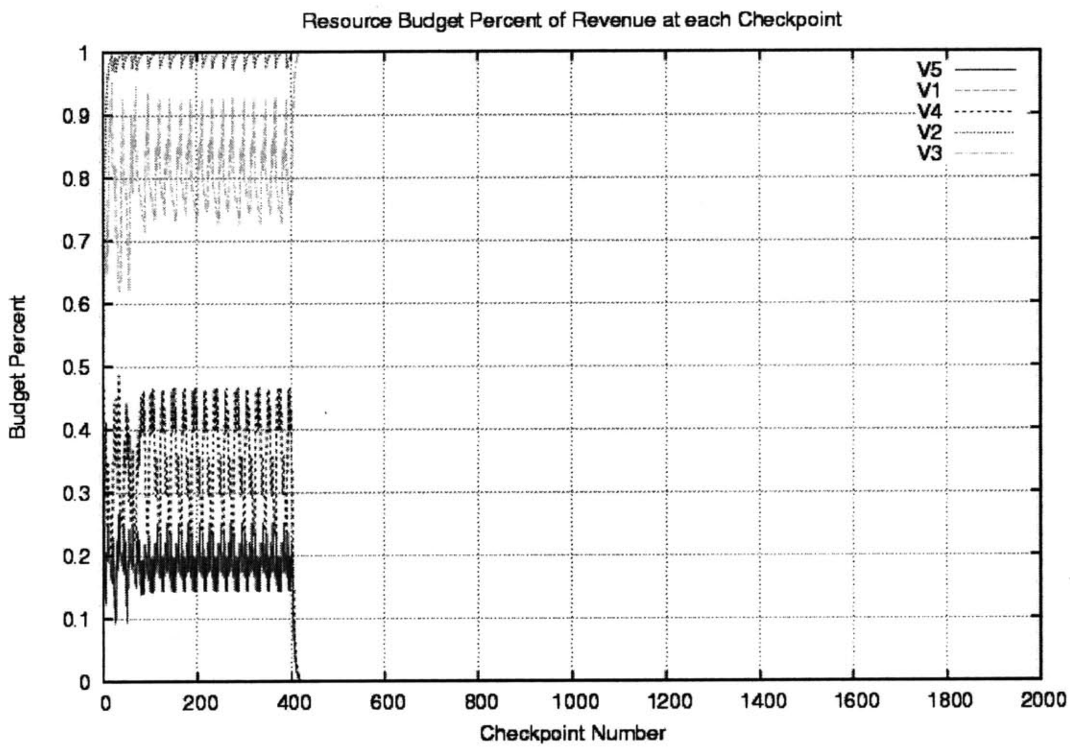


Figure 5-10: Resource budget percentage of revenue for market runtime handler with singular load change

(and when the exogenous load does not change and the market clearance mechanism is fair), feedback occurs in the form of a different resource allocation. After this point, resource budgets start to change in the opposite direction. This is the nature of the stable, yet fluctuating market.

5.5 Singular Load Change (Brute force)

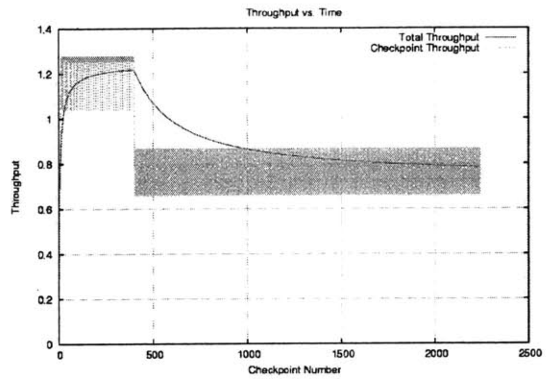
In order to test our aforementioned hypothesis that market clearance fairness provides feedback and thus stability, we reran the singular load change experiment with brute force runtime handlers. These runtime handlers search through the entire resource mapping space to pick the one which suits the particular policy the best. We see in figure 5-11 that the brute force market performs much better than the brute force static load balancer, and just slightly worse than the brute force dynamic load balancer. The market's throughput is approximately 91.98% of the dynamic load balancer's throughput.

runtime handler	throughput
static	0.439410
dynamic	0.849490
market(OFFSET=.2, OFFSET_SCALE=.25)	0.78134

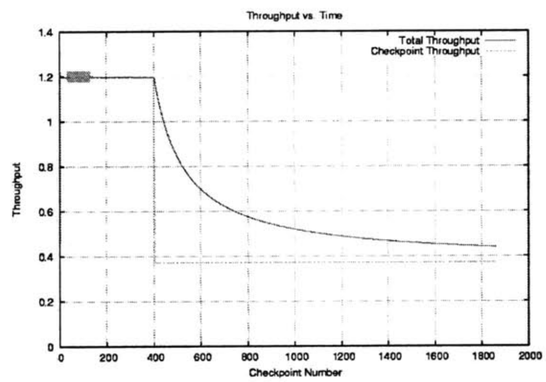
Figure 5-11: Results from the singular load change experiment (brute force)

Our hypothesis seems to be supported by the following figures. In figure 5-12(a), we see that throughput fluctuates after checkpoint 400 unlike figure 5-9(a). More importantly, we do not see the rapid movement to 0 and 1 in the resource budget plot of figure 5-13. The agents' resource budgets continue to fluctuate after the market shock, maintaining adaptability. Moreover, the movement to the new stable regime seems to occur by checkpoint 420, which is a relatively quick adaptation to the new market condition.

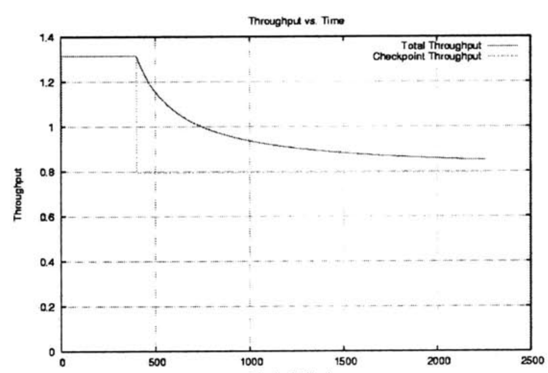
With the current system, we suspect that the greedy market clearance algorithm is much less robust than the brute force version (not taking into account its combinato-



(a) market (brute force)



(b) static (brute force)



(c) dynamic (brute force)

Figure 5-12: Throughput plots for the different runtime handlers responding to singular load change

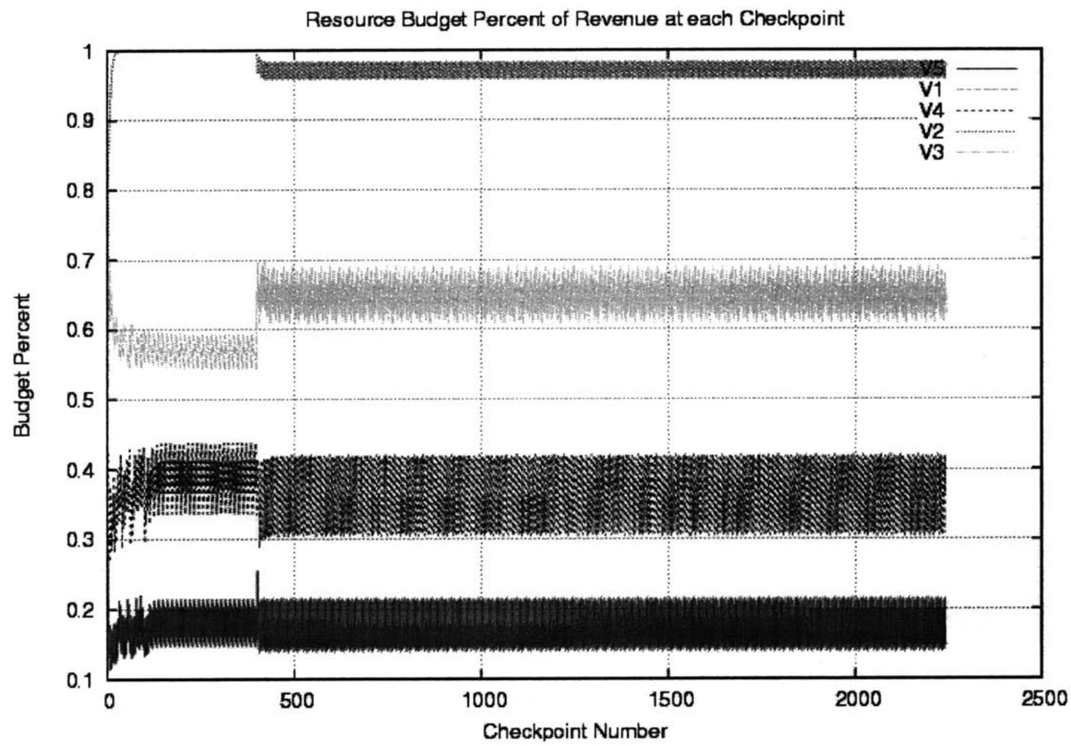


Figure 5-13: Resource budget percentage of revenue for market runtime handler with singular load change

rial overhead time). It just so happened that with our original processor speeds tested in section 5.2, the greedy market clearance algorithm was able to provide fairness. However, the speeds which resulted after the shock did not allow for this.

Chapter 6

Future Work

There still exist multiple facets of this research which we wish to explore further. First of all, we would like to make sure that our `StreamIt` simulation tool truly simulates `StreamIt` cluster computations accurately. One way to accomplish this is to run an actual computation on a cluster, prepare the corresponding stream graph and cluster graphs, and run the simulation. If the results are not satisfactory, we have outlined several possible reasons in section 4.1.3. We foresee the most significant of these to be ignoring overhead, network latency, and instruction-level effects. Network latency will be the easiest of the three to include in the simulation. This will simply involve setting up latency models analogous to processor load models, and including these effects when filters fire. If we can construct Java models of overhead-related calculations, and a unified method for estimating running times of the Java code, then we should be able to include overhead effects seamlessly with the rest of the simulation. Finer-grained simulation may also improve accuracy. Such additions could include a cache-model, or a more complete and accurate model of the operating system's process scheduler. Currently, we use a very simple round-robin scheduler estimate in our simulation.

There also exists significant latitude in changing the market structure of the dynamic load balancer. Currently, we only use one market agent algorithm at a time. We could certainly add heterogeneity to the system by including agents with different parameters at the same time. Moreover, we should explore different classes of agents entirely. These agent may be more complex in various ways. Firstly, they may use more variables than the current agents in determining how to distribute revenue. These agents may actually evolve in response to their own performance and the behaviors of others. Agents that perform poorly can be removed from the system in order to try out newer, possibly better policies. Another idea is to let agents store memories of their and other agents' past actions, and use this history information in making decisions.

A rather arbitrary constraint in the current system is that the agents must divide their leftover revenue (after deducting the resource budget) equally among its upstream neighbors. However, this seems like a poor decision if one of the upstream branches requires little computation compared to the other. One way to possibly change this policy would be to calculate the percentage of inputs which come from the different upstream neighbors. The agent could allot greater funds to those branches which provide fewer inputs, thus allowing this weaker branch to gain greater computational resources and produce data elements faster.

Another possible future avenue of research would involve changing the market clearance mechanism. There are probably many different ways of doing it than the way we have done. Moreover, we could foresee changing the market structure entirely. The current system is highly simplistic. Agents do not maintain currency or make any sort of trades among themselves, nor do we allow for explicit communication among them. Adding such features would certainly add interesting facets of complexity to the system.

Finally, the efficacy of this dynamic load-balancing mechanism will not be truly confirmed until it is actually implemented and tested on an actual `StreamIt` cluster computation. Such work would involve modifying the current cluster runtime libraries of `StreamIt` effectively adding a market layer on top of the current implementation.

Chapter 7

Conclusion

Throughout this paper, we have outlined a significant body of work on improving the runtime efficiency of `StreamIt` cluster computations while processing nodes are loaded to varying degrees. In order to easily study such runtime mechanisms, we have implemented a simulator for `StreamIt` which includes several different runtime handlers. We have built a graphical interface on top of this simulator to visually study the effects of the different runtime handling.

We have designed and implemented a market-like structure, modeling filters as agents which distribute revenue based on local metrics. These agents work together to adaptively configure the resource allocations. This market outperforms a static load-balancing technique which does not adapt to changes in load. It underperforms a more powerful technique, although we suspect markets to have lower overhead than such techniques in the real world.

In [5], the authors mention that dynamic load-balancing is an exciting area for future research. While their focus is more on using `StreamIt` for graphics processing, we nonetheless hope that this work provides at least a proof of concept for the utility of dynamic load-balancing.

While the work done in this thesis is specific to the `StreamIt` language, our hope is that it also provides an interesting case study in the more general field of complex adaptive systems. Our market-like system includes multiple agents acting (mostly) on their own behalf, with incomplete knowledge of the entire system, yet they are able to interact in such a way that the entire system performs more efficiently. Because each filter's throughput performance depends on the performance of the other filters, agents must cooperate. They must balance the desire for local optimization with the needs of the other agents.

Appendix A

StreamIt Simulator Javadocs

A.1 Package runtime.market

Package Contents *Page*

A.1.1 Interface Agent

Agent is the interface for all types of Agents

Declaration

public interface Agent

All known subinterfaces

FilterAgent (in A.1.4, page 71), ResourceAgent (in A.1.7, page 84), AdaptiveFilterAgent (in A.1.3, page 69)

All classes known to implement interface

FilterAgent (in A.1.4, page 71), ResourceAgent (in A.1.7, page 84)

Method summary

notifyCheckpoint(double)

Methods

- **notifyCheckpoint**
  `void` **notifyCheckpoint**`( double t )`

A.1.2 Interface MarketRuntimeHandler.AgentFunction

An AgentFunction takes in an agent and returns a T.

Declaration

private static interface MarketRuntimeHandler.AgentFunction

Method summary

getValue(FilterAgent)

Methods

- **getValue**
  `java.lang.Object` **getValue(** `FilterAgent a` **)**

A.1.3 Class AdaptiveFilterAgent

AdaptiveFilterAgent is a FilterAgent which makes distribution decisions based on how it has done in the past. A Filter F is in one of 3 states: 1. Working (t = W) 2. Waiting for input (t = dW) 3. Waiting for downstream buffers (t = bW) Let T be the duration of the previous checkpoint, then: idleTime = T - W If idleTime is very low, spend less on proc Otherwise: If dW is high, spend less on proc (give more money to upstream) If bW is high, spend less on proc

Declaration

public class AdaptiveFilterAgent
extends runtime.market.FilterAgent (in A.1.4, page 71)

Field summary

OFFSET
OFFSET_SCALE

Constructor summary

AdaptiveFilterAgent(MarketRuntimeHandler, FilterVertex, double, double) Construct the AdaptiveFilterAgent

Method summary

distributeRevenue() This method represents implementation of the strategy of the agent.

Fields

- private final double **OFFSET**
- private final double **OFFSET_SCALE**

Constructors

- **AdaptiveFilterAgent**
  `public` **AdaptiveFilterAgent**`( MarketRuntimeHandler` **h**`, sim.FilterVertex` **fv**`, double` **offset**`, double` **offset_scale** `)`
  - **Description**
    Construct the AdaptiveFilterAgent

Methods

- **distributeRevenue**
  `protected void` **distributeRevenue**`( )`
  - **Description**
    This method represents implementation of the strategy of the agent. It allocates the agent's revenue among who it buys resources from.

Members inherited from class `runtime.market.FilterAgent` (in A.1.4, page 71)

- `public static void` **calcDataPrice**`( FilterAgent` **fav** `)`
- `private void` **calcPrices**`( )`
- `private boolean` **checkChildren**`( )`
- `protected` **checkpointTime**
- `private` **childrenCalc**
- `private void` **clearCheckpointVars**`( )`
- `private` **dataCost**
- `protected` **dataPrices**
- `private` **dataRev**
- `protected` **dataWaitTime**
- `protected abstract void` **distributeRevenue**`( )`
- `private static void` **doSale**`( FilterAgent` **firer**`, FilterAgent` **buyer** `)`
- `protected final` **filterVertex**
- `protected FilterAgent` **getAgent**`( sim.FilterVertex` **firer** `)`
- `public double` **getBudgetPct**`( )`
- `public double` **getCheckpointTime**`( )`
- `private double` **getDataPrice**`( FilterAgent` **firer** `)`
- `public Map` **getDataPrices**`( )`
- `public double` **getDataWaitTime**`( )`
- `public FilterVertex` **getFilterVertex**`( )`
- `public double` **getIdleTime**`( )`
- `public double` **getNumberPushed**`( )`

- public double **getResourceBudget**( )
- protected double **getRevenue**( )
- private **lastCheckpointTime**
- private **lastDataWaitTime**
- private **lastPush**
- private **lastWorkTime**
- public void **notifyCheckpoint**(double **t**)
- public void **notifyDataWait**(double **time**)
- public void **notifyDoneCheckpoint**( )
- public void **notifyFire**(double **time**)
- public void **notifyInject**(sim.FilterVertex **firer**)
- private **numInjected**
- private **numPush**
- protected **oldDataPrices**
- protected **oldResourceMoney**
- private **resourceMoney**
- private final **runtime**
- private void **saveCheckpointVars**( )
- public String **toString**( )
- protected **workTime**

A.1.4 Class FilterAgent

The FilterAgent class represents agents acting on behalf of the various filters in the StreamIt computation. Each of these agents decides how to allocate its revenue among incoming data elements and resource costs.

Declaration

public abstract class FilterAgent
extends java.lang.Object
implements Agent, sim.FilterListener

All known subclasses

AdaptiveFilterAgent (in A.1.3, page 69)

Field summary

checkpointTime
childrenCalc Number of children calculated already in this price calculation
dataCost
dataPrices Maps incoming vertices' agents into prices
dataRev
dataWaitTime Time waiting for input during checkpoint
filterVertex The vertex this agent represents
lastCheckpointTime
lastDataWaitTime
lastPush
lastWorkTime
numInjected Number of elements injected into this filter during checkpoint
numPush Number of elements pushed during checkpoint
oldDataPrices The old value of dataPrices (previous checkpoint)
oldResourceMoney The old value of resourceMoney (previous checkpoint)
resourceMoney How much to allocate to processor
runtime The global market
workTime Time worked during checkpoint

Constructor summary

FilterAgent(MarketRuntimeHandler, FilterVertex) Construct a FilterAgent

Method summary

calcDataPrice(FilterAgent) Calculate the data prices for this economy
calcPrices() Recursively calculate data prices for the different transactions.
checkChildren() Check to see if children's data prices have been calculated yet
clearCheckpointVars() Clear variables which accumulate during each checkpoint
distributeRevenue() This method represents implementation of the strategy of the agent.
doSale(FilterAgent, FilterAgent)
getAgent(FilterVertex) Return the agent working on behalf of the given FilterVertex in this economy
getBudgetPct() Return what percentage of this agent's income goes to processing power
getCheckpointTime()
getDataPrice(FilterAgent) Return the price this agent has set for the data from firer
getDataPrices() Return aAn unmodifiable view of the data prices for this agent's children
getDataWaitTime() Return how long this filter was waiting for input during the last checkpoint
getFilterVertex() Return the FilterVertex this agent works for

getIdleTime() Return how long this filter was idle during the last checkpoint
getNumberPushed() Return how many elements this filter pushed out during the last checkpoint
getResourceBudget() Return how much this agent will spend on processing power
getRevenue() Get average revenue per firing of the filter.
notifyCheckpoint(double)
notifyDataWait(double) Increment dataWaitTime
notifyDoneCheckpoint()
notifyFire(double) Increment numPush and workTime at each firing
notifyInject(FilterVertex) Increment numInjected
saveCheckpointVars() Copy previous checkpoint values to "old" variables
toString()

Fields

- private final MarketRuntimeHandler **runtime**
  - The global market
- protected final sim.FilterVertex **filterVertex**
  - The vertex this agent represents
- protected java.util.Map **dataPrices**
  - Maps incoming vertices' agents into prices
- protected java.util.Map **oldDataPrices**
  - The old value of dataPrices (previous checkpoint)
- protected double **oldResourceMoney**
  - The old value of resourceMoney (previous checkpoint)
- private int **childrenCalc**
  - Number of children calculated already in this price calculation
- private double **resourceMoney**
  - How much to allocate to processor
- private int **numPush**
  - Number of elements pushed during checkpoint
- private int **numInjected**
  - Number of elements injected into this filter during checkpoint
- protected double **workTime**
  - Time worked during checkpoint
- protected double **dataWaitTime**

– Time waiting for input during checkpoint

- private double **dataRev**
- private double **dataCost**
- protected double **checkpointTime**
- private double **lastCheckpointTime**
- private double **lastWorkTime**
- private double **lastDataWaitTime**
- private int **lastPush**

Constructors

- **FilterAgent**
  `public` **FilterAgent**(`MarketRuntimeHandler` **h**, `sim.FilterVertex` **fv**)
  - **Description**
    Construct a FilterAgent
  - **Parameters**
    - `h` – The market handling the resource management
    - `fv` – The FilterVertex this agent works for

Methods

- **calcDataPrice**
  `public static void` **calcDataPrice**(`FilterAgent` **fav**)
  - **Description**
    Calculate the data prices for this economy
- **calcPrices**
  `private void` **calcPrices**( )
  - **Description**
    Recursively calculate data prices for the different transactions. Sets the dataPrices and resourceMoney fields of each filter agent.
- **checkChildren**
  `private boolean` **checkChildren**( )
  - **Description**
    Check to see if children's data prices have been calculated yet
- **clearCheckpointVars**
  `private void` **clearCheckpointVars**( )

– **Description**
Clear variables which accumulate during each checkpoint

- **distributeRevenue**
`protected abstract void` **distributeRevenue**`( )`

  – **Description**
  This method represents implementation of the strategy of the agent. It allocates the agent's revenue among who it buys resources from.

- **doSale**
`private static void` **doSale**`( FilterAgent` **firer**`, FilterAgent` **buyer** `)`

- **getAgent**
`protected FilterAgent` **getAgent**`( sim.FilterVertex` **firer** `)`

  – **Description**
  Return the agent working on behalf of the given FilterVertex in this economy

- **getBudgetPct**
`public double` **getBudgetPct**`( )`

  – **Description**
  Return what percentage of this agent's income goes to processing power

- **getCheckpointTime**
`public double` **getCheckpointTime**`( )`

- **getDataPrice**
`private double` **getDataPrice**`( FilterAgent` **firer** `)`

  – **Description**
  Return the price this agent has set for the data from firer

- **getDataPrices**
`public java.util.Map` **getDataPrices**`( )`

  – **Description**
  Return aAn unmodifiable view of the data prices for this agent's children

- **getDataWaitTime**
`public double` **getDataWaitTime**`( )`

  – **Description**
  Return how long this filter was waiting for input during the last checkpoint

- **getFilterVertex**
`public sim.FilterVertex` **getFilterVertex**`( )`

  – **Description**
  Return the FilterVertex this agent works for

- **getIdleTime**
  `public double` **getIdleTime**( )

  - **Description**

    Return how long this filter was idle during the last checkpoint

- **getNumberPushed**
  `public double` **getNumberPushed**( )

  - **Description**

    Return how many elements this filter pushed out during the last checkpoint

- **getResourceBudget**
  `public double` **getResourceBudget**( )

  - **Description**

    Return how much this agent will spend on processing power

- **getRevenue**
  `protected double` **getRevenue**( )

  - **Description**

    Get average revenue per firing of the filter. This calculation depends on the data prices set by the downstream filters, and this filter's push value.

- **notifyCheckpoint**
  `void` **notifyCheckpoint**(`double t`)

- **notifyDataWait**
  `public void` **notifyDataWait**(`double` **time**)

  - **Description**

    Increment dataWaitTime

- **notifyDoneCheckpoint**
  `public void` **notifyDoneCheckpoint**( )

- **notifyFire**
  `public void` **notifyFire**(`double` **time**)

  - **Description**

    Increment numPush and workTime at each firing

- **notifyInject**
  `public void` **notifyInject**(`sim.FilterVertex` **firer**)

  - **Description**

    Increment numInjected

- **saveCheckpointVars**
  `private void` **saveCheckpointVars**( )

- **Description**

  Copy previous checkpoint values to "old" variables

- **toString**
  `public java.lang.String` **toString( )**

A.1.5 Class MarketRuntimeHandler

The MarketRuntimeHandler implements RuntimeHandler. It works by taking the resource budgets of the individual filter agents, and clearing the market for global efficiency.

Declaration

public class MarketRuntimeHandler
extends java.lang.Object
implements runtime.RuntimeHandler

Field summary

checkpointNum
checkpointNums
checkpointThroughputHistory
desc The computation descriptor
filterAgents The agents for filters
genericAgentDataHistory
lastOutputs
lastOutputTime
loadHistory
OFFSET Offset param for adaptive filter agents
OFFSET_SCALE Offset_scale param for adaptive filter agents
outFile
plotter
resourceAgents The agents for the resources (not currently used)
resourceCostHistory
resourceMap The current resource allocation
rht
sim The Simulator
throughputHistory
WINDOW

Constructor summary

MarketRuntimeHandler(Simulator, StreamItComputationDescriptor, double, double, Simulator.rhType) Creates the runtime handler for the given computation

Method summary

appendDataFile(double)
checkpoint(double) Clears the market, setting up the resource mapping
clearMarket(double) Use the resource allotment of each filter agent to create the resource mapping
createAgents() Allocate private the resource maps
doPlots() Actually write the appropriate octave scripts to plot to the file
evaluateMapping(Map, double) A heuristic evaluation of a mapping based on market balancing balancing.
evaluateMappingBruteForce(Map, double) A heuristic evaluation of a mapping based on market balancing balancing.
finished() Write the plots and close the file
genericDataUpdate(double) Call at each checkpoint to update data for plots
getAvg(int)
getBestMap(double) Get the best mapping based on the one which evaluates to the lowest score
getBestMapBruteForce(double) Get the best mapping based on the one which evaluates to the lowest score (brute force)
getBestPCV(double, FilterVertex, Map)
getFilterAgent(FilterVertex) Get the FilterAgent associated with firer in this market
getFilterAgents() Get all FilterAgents in the market
getMappings() Return a set of all possible resource mappings
getSpeedInMap(Map, FilterAgent, double) Get the predicted speed of the FilterAgent in the given at the given time
loadHistory(double) Update load history
notifyCheckpoint(double) Notify the agents of a checkpoint
notifyDoneCheckpoint()
orderAgents() Order the vertices in descending order of resource budget for their associated agent
resourceCostHistory(double) Update resource cost history
runMarket(double) Runs the market clearing mechanism at the given time
throughputHistory(double)
writeSummary(PrintStream, int) Write octave code to display a throughput summary across repititions of the simulation

Fields

- private final sim.desc.StreamItComputationDescriptor **desc**
  - The computation descriptor
- private final sim.Simulator **sim**
  - The Simulator
- private java.util.Map **resourceMap**
  - The current resource allocation

- private final java.util.Map **filterAgents**
  - The agents for filters
- private final java.util.Map **resourceAgents**
  - The agents for the resources (not currently used)
- private final runtime.Plotter **plotter**
- private final java.io.PrintWriter **outFile**
- private final java.util.List **checkpointNums**
- private final java.util.List **throughputHistory**
- private final java.util.List **checkpointThroughputHistory**
- private final java.util.Map **resourceCostHistory**
- private final java.util.Map **loadHistory**
- private final java.util.Set **genericAgentDataHistory**
- private final double **OFFSET**
  - Offset param for adaptive filter agents
- private final double **OFFSET_SCALE**
  - Offset_scale param for adaptive filter agents
- private final sim.Simulator.rhType **rht**
- private int **checkpointNum**
- private long **lastOutputs**
- private double **lastOutputTime**
- private static int **WINDOW**

Constructors

- **MarketRuntimeHandler**
  `public` **MarketRuntimeHandler**`( sim.Simulator` **theSim**`, sim.desc.StreamItComputationDes`
  theDesc`, double` **offset**`, double` **offset_scale**`, sim.Simulator.rhType` **rht** `)`
  - **Description**
    Creates the runtime handler for the given computation
  - **Parameters**
    * `descriptor` – The computation descriptor

Methods

- **appendDataFile**
  `private void appendDataFile( double t )`

- **checkpoint**
  `public java.util.Map checkpoint( double time )`

  - **Description**
    Clears the market, setting up the resource mapping

- **clearMarket**
  `private void clearMarket( double t )`

  - **Description**
    Use the resource allotment of each filter agent to create the resource mapping

- **createAgents**
  `private void createAgents( )`

  - **Description**
    Allocate private the resource maps

- **doPlots**
  `private void doPlots( )`

  - **Description**
    Actually write the appropriate octave scripts to plot to the file

- **evaluateMapping**
  `private double evaluateMapping( java.util.Map map, double t )`

  - **Description**
    A heuristic evaluation of a mapping based on market balancing balancing.
  - **Parameters**
    * `map` – The mapping of filter to resource
  - **Returns** – a score of how good the mapping is (the lower, the better)

- **evaluateMappingBruteForce**
  `private double evaluateMappingBruteForce( java.util.Map map, double t )`

  - **Description**
    A heuristic evaluation of a mapping based on market balancing balancing.
  - **Parameters**
    * `map` – The mapping of filter to resource
  - **Returns** – a score of how good the mapping is (the lower, the better)

- **finished**
  `public void finished( )`

– **Description**
  Write the plots and close the file

- **genericDataUpdate**
  `private void` **genericDataUpdate**`( double` **t** `)`
  – **Description**
    Call at each checkpoint to update data for plots
  – **Parameters**
    * `t` – The time

- **getAvg**
  `private double` **getAvg**`( int` **window** `)`

- **getBestMap**
  `private java.util.Map` **getBestMap**`( double` **t** `)`
  – **Description**
    Get the best mapping based on the one which evaluates to the lowest score

- **getBestMapBruteForce**
  `private java.util.Map` **getBestMapBruteForce**`( double` **t** `)`
  – **Description**
    Get the best mapping based on the one which evaluates to the lowest score (brute force)

- **getBestPCV**
  `private sim.PCVertex` **getBestPCV**`( double` **t**`, sim.FilterVertex` **fv**`, java.util.Map` **startMap** `)`

- **getFilterAgent**
  `public FilterAgent` **getFilterAgent**`( sim.FilterVertex` **firer** `)`
  – **Description**
    Get the FilterAgent associated with firer in this market

- **getFilterAgents**
  `public java.util.Collection` **getFilterAgents**`( )`
  – **Description**
    Get all FilterAgents in the market

- **getMappings**
  `private java.util.Set` **getMappings**`( )`
  – **Description**
    Return a set of all possible resource mappings
  – **Returns** –

- **getSpeedInMap**
  `private double` **getSpeedInMap**`( java.util.Map` **map**`, FilterAgent` **a**`, double` **t** `)`

  – **Description**

    Get the predicted speed of the FilterAgent in the given at the given time

  – **Parameters**

    * `map` – The resource map
    * `a` – The agent
    * `t` – The time (used to look at processor load)

  – **Returns** – the predicted time

- **loadHistory**
  `private void` **loadHistory**`( double` **t** `)`

  – **Description**

    Update load history

  – **Parameters**

    * `t` – The time

- **notifyCheckpoint**
  `private void` **notifyCheckpoint**`( double` **t** `)`

  – **Description**

    Notify the agents of a checkpoint

- **notifyDoneCheckpoint**
  `private void` **notifyDoneCheckpoint**`( )`

- **orderAgents**
  `private java.util.List` **orderAgents**`( )`

  – **Description**

    Order the vertices in descending order of resource budget for their associated agent

- **resourceCostHistory**
  `private void` **resourceCostHistory**`( double` **t** `)`

  – **Description**

    Update resource cost history

  – **Parameters**

    * `t` – The time

- **runMarket**
  `private void` **runMarket**`( double` **time** `)`

- **Description**

  Runs the market clearing mechanism at the given time
- **Parameters**
  * `time` – The time we are clearing the market

- **throughputHistory**
  `private void` **throughputHistory**`( double` **t** `)`

- **writeSummary**
  `public void` **writeSummary**`( java.io.PrintStream` **file**`, int` **rep** `)`
  - **Description**

    Write octave code to display a throughput summary across repititions of the simulation

A.1.6 Class MarketRuntimeHandler.AgentData

Inner class which helps collect agent data for plots at each checkpoint

Declaration

private class MarketRuntimeHandler.AgentData
extends java.lang.Object

Field summary

dataMap All the data
func The AgentFunction which gets the data
name
yAxis

Constructor summary

MarketRuntimeHandler.AgentData(String, String, MarketRuntime-Handler.AgentFunction) Construct an AgentData object

Method summary

getData() Return an unmodifiable view of the data
getName()
getYAxis()
update() Call at each checkpoint to update the data

Fields

- private final java.lang.String **name**
- private final java.lang.String **yAxis**

- private final MarketRuntimeHandler.AgentFunction **func**
  - The AgentFunction which gets the data
- private final java.util.Map **dataMap**
  - All the data

Constructors

- **MarketRuntimeHandler.AgentData**
  `public` **MarketRuntimeHandler.AgentData(** `java.lang.String` **name,** `java.lang.String` **yAxis,** `MarketRuntimeHandler.AgentFunction` **function)**
  - **Description**
    Construct an AgentData object
  - **Parameters**
    * `name` – Name of the plot
    * `yAxis` – Text for the y axis label
    * `function` – The AgentFunction

Methods

- **getData**
  `public java.util.Map` **getData( )**
  - **Description**
    Return an unmodifiable view of the data
- **getName**
  `public java.lang.String` **getName( )**
- **getYAxis**
  `public java.lang.String` **getYAxis( )**
- **update**
  `public void` **update( )**
  - **Description**
    Call at each checkpoint to update the data

A.1.7 Class ResourceAgent

A ResourceAgent represents a processor in the marketplace. Currently, it does nothing.

Declaration

public class ResourceAgent
extends java.lang.Object
implements Agent

Field summary

PCVertex

Constructor summary

ResourceAgent(PCVertex)

Method summary

notifyCheckpoint(double)

Fields

- private final sim.PCVertex **PCVertex**

Constructors

- **ResourceAgent**
  `public` **ResourceAgent**`( sim.PCVertex` **pcv** `)`

Methods

- **notifyCheckpoint**
  `void` **notifyCheckpoint**`( double` **t** `)`

A.2 Package runtime

A.2.1 Interface RuntimeHandler

The RuntimeHandler determines the mapping of StreamIt filters to PCs in the cluster

Declaration

public interface RuntimeHandler

All known subinterfaces

MarketRuntimeHandler (in A.1.5, page 77), StaticLoadBalancer (in A.2.4, page 90), StaticRuntimeHandler (in A.2.5, page 92), DynamicLoadBalancer (in A.2.2, page 86)

All classes known to implement interface

MarketRuntimeHandler (in A.1.5, page 77), StaticLoadBalancer (in A.2.4, page 90), StaticRuntimeHandler (in A.2.5, page 92), DynamicLoadBalancer (in A.2.2, page 86)

Method summary

checkpoint(double) Creates the initial resource allocation mapping
finished()
writeSummary(PrintStream, int)

Methods

- **checkpoint**
  `java.util.Map` **checkpoint**`( double` **time** `)`
  - **Description**
    Creates the initial resource allocation mapping
  - **Parameters**
    * `computation` – The StreamIt computation description
  - **Returns** –
- **finished**
  `void` **finished**`( )`
- **writeSummary**
  `void` **writeSummary**`( java.io.PrintStream` **file**`, int` **rep** `)`

A.2.2 Class DynamicLoadBalancer

Finds the near-optimal resource allocation.

Declaration

public class DynamicLoadBalancer
extends java.lang.Object
implements RuntimeHandler

Field summary

bestMapping The resource map that is most statically load-balanced
desc The StreamIt computation descriptor

Constructor summary

DynamicLoadBalancer(StreamItComputationDescriptor)

Method summary

checkpoint(double)
evaluateMapping(Map, double) A heuristic evaluation of a mapping based on load balancing.
finished()
getBestMap(double) Get the best mapping based on the one which evaluates to the lowest score
getBestPCV(FilterVertex, Map, double)
getMappings() Return all possible resource mappings
getTotalWork(Map, PCVertex) Get total work needed in the given resource mapping at the given processor
orderFilters() Order the vertices in descending order of work
writeSummary(PrintStream, int)

Fields

- private final sim.desc.StreamItComputationDescriptor **desc**
  - The StreamIt computation descriptor
- private java.util.Map **bestMapping**
  - The resource map that is most statically load-balanced

Constructors

- **DynamicLoadBalancer**
  `public` **DynamicLoadBalancer**`( sim.desc.StreamItComputationDescriptor` **desc**
  `)`

Methods

- **checkpoint**
  `java.util.Map` **checkpoint**`( double` **time** `)`
  - **Description copied from RuntimeHandler (in A.2.1, page 85)**
    Creates the initial resource allocation mapping
  - **Parameters**
    * `computation` – The StreamIt computation description
  - **Returns** –

- **evaluateMapping**
  `private double` **evaluateMapping(** `java.util.Map` **map**, `double` **t**)

  - **Description**

    A heuristic evaluation of a mapping based on load balancing.

  - **Parameters**

    * `map` – The mapping of filter to resource

  - **Returns** – a score of how good the mapping is (the lower, the better)

- **finished**
  `void` **finished(**)

- **getBestMap**
  `private java.util.Map` **getBestMap(** `double` **t**)

  - **Description**

    Get the best mapping based on the one which evaluates to the lowest score

- **getBestPCV**
  `private sim.PCVertex` **getBestPCV(** `sim.FilterVertex` **fv**, `java.util.Map` **startMap**, `double` **t**)

- **getMappings**
  `private java.util.Set` **getMappings(**)

  - **Description**

    Return all possible resource mappings

- **getTotalWork**
  `private double` **getTotalWork(** `java.util.Map` **map**, `sim.PCVertex` **v**)

  - **Description**

    Get total work needed in the given resource mapping at the given processor

- **orderFilters**
  `private java.util.List` **orderFilters(**)

  - **Description**

    Order the vertices in descending order of work

- **writeSummary**
  `void` **writeSummary(** `java.io.PrintStream` **file**, `int` **rep**)

A.2.3 Class Plotter

The Plotter class is useful for writing octave scripts from java data structures. When executing the octave script, the output is plots.

Declaration

public class Plotter
extends java.lang.Object

Field summary

fileName Name of the file to output the plot to
out The output stream

Constructor summary

Plotter(PrintWriter, String) Construct a Plotter

Method summary

octaveVector(List) Produce a vector in octave format given a list of data
writePlot(String, String, String, List, Map) Write a plotting script

Fields

- private final java.io.PrintWriter **out**
  - The output stream
- private final java.lang.String **fileName**
  - Name of the file to output the plot to

Constructors

- **Plotter**
  `public` **Plotter**`( java.io.PrintWriter` **writer**, `java.lang.String` **Name**)
  - **Description**
    Construct a Plotter
  - **Parameters**
    * `writer` – The stream to write the script to
    * `Name` – File the script creates when executed

Methods

- **octaveVector**
  `private static java.lang.String` **octaveVector**`( java.util.List` **data**)
  - **Description**
    Produce a vector in octave format given a list of data
- **writePlot**
  `public void` **writePlot**`( java.lang.String` **plotName**, `java.lang.String` **xAxis**, `java.lang.String` **yAxis**, `java.util.List` **xData**, `java.util.Map` **yData**)

- **Description**

  Write a plotting script
- **Parameters**
  * `plotName` – Name of the plot
  * `xAxis` – Label for x axis
  * `yAxis` – Label for y axis
  * `xData` – Data points on the x axis
  * `yData` – Mapping which contains a collection of y axis data points in the values. The key is used for the legend

A.2.4 Class StaticLoadBalancer

Finds the near-optimal static resource allocation. Models what the StreamIt compiler will do to optimize the cluster computation.

Declaration

public class StaticLoadBalancer
extends java.lang.Object
implements RuntimeHandler

Field summary

bestMapping The resource map that is most statically load-balanced
desc The StreamIt computation descriptor

Constructor summary

StaticLoadBalancer(StreamItComputationDescriptor)

Method summary

checkpoint(double)
evaluateMapping(Map) A heuristic evaluation of a mapping based on load balancing.
finished()
getBestMap() Get the best mapping based on the one which evaluates to the lowest score
getBestMapBruteForce() Search through all possible mappings to find the best one, according to the evaluation function
getBestPCV(FilterVertex, Map)
getMappings() Return all possible resource mappings
getTotalWork(Map, PCVertex) Get total work needed in the given resource mapping at the given processor
orderFilters() Order the vertices in descending order of work
writeSummary(PrintStream, int)

Fields

- private final sim.desc.StreamItComputationDescriptor **desc**
  - The StreamIt computation descriptor
- private java.util.Map **bestMapping**
  - The resource map that is most statically load-balanced

Constructors

- **StaticLoadBalancer**
  `public StaticLoadBalancer( sim.desc.StreamItComputationDescriptor desc )`

Methods

- **checkpoint**
  `java.util.Map checkpoint( double time )`
  - **Description copied from RuntimeHandler (in A.2.1, page 85)**
    Creates the initial resource allocation mapping
  - **Parameters**
    - `computation` – The StreamIt computation description
  - **Returns** –
- **evaluateMapping**
  `private double evaluateMapping( java.util.Map map )`
  - **Description**
    A heuristic evaluation of a mapping based on load balancing.
  - **Parameters**
    - `map` – The mapping of filter to resource
  - **Returns** – a score of how good the mapping is (the lower, the better)
- **finished**
  `void finished( )`
- **getBestMap**
  `private java.util.Map getBestMap( )`
  - **Description**
    Get the best mapping based on the one which evaluates to the lowest score
- **getBestMapBruteForce**
  `private java.util.Map getBestMapBruteForce( )`
  - **Description**
    Search through all possible mappings to find the best one, according to the evaluation function

- **getBestPCV**
  `private sim.PCVertex` **getBestPCV**`( sim.FilterVertex` **fv**`, java.util.Map` **startMap** `)`

- **getMappings**
  `private java.util.Set` **getMappings**`( )`

  - **Description**
    Return all possible resource mappings

- **getTotalWork**
  `private double` **getTotalWork**`( java.util.Map` **map**`, sim.PCVertex` **v** `)`

  - **Description**
    Get total work needed in the given resource mapping at the given processor

- **orderFilters**
  `private java.util.List` **orderFilters**`( )`

  - **Description**
    Order the vertices in descending order of work

- **writeSummary**
  `void` **writeSummary**`( java.io.PrintStream` **file**`, int` **rep** `)`

A.2.5 Class StaticRuntimeHandler

A simple RuntimeHandler. Choose the mapping arbitrarily.

Declaration

public class StaticRuntimeHandler
extends java.lang.Object
implements RuntimeHandler

Field summary

desc

Constructor summary

StaticRuntimeHandler(StreamItComputationDescriptor)

Method summary

checkpoint(double)
finished()
writeSummary(PrintStream, int)

Fields

- private final sim.desc.StreamItComputationDescriptor **desc**

Constructors

- **StaticRuntimeHandler**
  `public` **StaticRuntimeHandler**`( sim.desc.StreamItComputationDescriptor` **descriptor** `)`

Methods

- **checkpoint**
  `java.util.Map` **checkpoint**`( double` **time** `)`
  - **Description copied from RuntimeHandler (in A.2.1, page 85)**
    Creates the initial resource allocation mapping
  - **Parameters**
    * `computation` – The StreamIt computation description
  - **Returns** –
- **finished**
  `void` **finished**`( )`
- **writeSummary**
  `void` **writeSummary**`( java.io.PrintStream` **file**, `int` **rep** `)`

A.3 Package gui

Package Contents *Page*

Classes

A.3.1 Class SimulatorGUI

SimulatorGUI is responsible for visually displaying the StreamIt simulation. Uses VisualizationViewer from the JUNG library to display graphs. Code based off of the Swing Tutorial

Declaration

public class SimulatorGUI
extends java.lang.Object
implements sim.SimulationListener

Field summary

frame The main frame in the GUI
LOOKANDFEEL
OFFSET
sim The simulation
throughputLabel The textual throughput display
vv View of the graph

Constructor summary

SimulatorGUI(Simulator)

Method summary

createAndShowGUI() Create the GUI and show it.
createComponents() Creates a pane with the VisualizationViewer inside
initLookAndFeel()
invoke(Runnable)
invokeLater(Runnable)
notifyExit()
notifyOutput(long, double)
notifyRun()
setupLayout(Layout, Dimension)

Fields

- private final sim.Simulator **sim**
  - The simulation
- private javax.swing.JFrame **frame**
  - The main frame in the GUI
- private edu.uci.ics.jung.visualization.VisualizationViewer **vv**
  - View of the graph
- private javax.swing.JLabel **throughputLabel**
  - The textual throughput display
- static final java.lang.String **LOOKANDFEEL**
- private static final int **OFFSET**

Constructors

- **SimulatorGUI**
  `public` **SimulatorGUI**`( sim.Simulator` **sim** `)`

Methods

- **createAndShowGUI**
  `private void` **createAndShowGUI**`( )`
  - **Description**
    Create the GUI and show it. For thread safety, this method should be invoked from the event-dispatching thread.

- **createComponents**
  `public java.awt.Component` **createComponents**`( )`
  - **Description**
    Creates a pane with the VisualizationViewer inside

- **initLookAndFeel**
  `private static void` **initLookAndFeel**`( )`

- **invoke**
  `private static void` **invoke**`( java.lang.Runnable` **run** `)`

- **invokeLater**
  `private static void` **invokeLater**`( java.lang.Runnable` **run** `)`

- **notifyExit**
  `void` **notifyExit**`( )`
  - **Description copied from sim.SimulationListener** (**in A.6.2, page 117**)
    Called when the simulation exits

- **notifyOutput**
  `void` **notifyOutput**`( long` **outputs**`, double` **lastOutputTime** `)`
  - **Description copied from sim.SimulationListener** (**in A.6.2, page 117**)
    Called when the simulation produces more outputs (the final node pushed data out)

- **notifyRun**
  `void` **notifyRun**`( )`
  - **Description copied from sim.SimulationListener** (**in A.6.2, page 117**)
    Called when the simulation begins

- **setupLayout**
  `private void` **setupLayout**`( edu.uci.ics.jung.visualization.Layout` **l**`, java.awt.Dimension` **d** `)`

A.3.2 Class SimulatorGUI.ToolTip

Inner class to display tooltip info about vertices.

Declaration

private class SimulatorGUI.ToolTip
extends java.lang.Object
implements edu.uci.ics.jung.graph.decorators.ToolTipFunction

Constructor summary

SimulatorGUI.ToolTip()

Method summary

getToolTipText(Edge)
getToolTipText(MouseEvent)
getToolTipText(Vertex)

Constructors

- **SimulatorGUI.ToolTip**
  `private` **SimulatorGUI.ToolTip**`( )`

Methods

- **getToolTipText**
  `java.lang.String` **getToolTipText**`( edu.uci.ics.jung.graph.Edge` **arg0** `)`
- **getToolTipText**
  `public java.lang.String` **getToolTipText**`( java.awt.event.MouseEvent` **event**
  `)`
- **getToolTipText**
  `java.lang.String` **getToolTipText**`( edu.uci.ics.jung.graph.Vertex` **arg0** `)`

A.4 Package sim.parse

Package Contents *Page*

A.4.1 Class SimulatorOptionsParser

JCommando generated parser class.

Declaration

public abstract class SimulatorOptionsParser
extends org.jcommando.JCommandParser

All known subclasses

Simulator.SimulatorOptions (in A.6.8, page 133)

Constructor summary

SimulatorOptionsParser() JCommando generated constructor.

Method summary

createExecuteGrouping() Generate the grouping for the 'execute' command.
doExecute() Called by parser to perform the 'execute' command.
setCli() Called by parser to set the 'cli' property.
setCluster_file(String) Called by parser to set the 'cluster_file' property.
setGui_delay(long) Called by parser to set the 'gui_delay' property.
setGui() Called by parser to set the 'gui' property.
setHelp() Called by parser to set the 'help' property.
setNumber_firings(long) Called by parser to set the 'number_firings' property.
setOffset_scale(String) Called by parser to set the 'offset_scale' property.
setOffset(String) Called by parser to set the 'offset' property.
setOut_file(String) Called by parser to set the 'out_file' property.
setRep(long) Called by parser to set the 'rep' property.
setRuntime_handler(String) Called by parser to set the 'runtime_handler' property.
setStream_file(String) Called by parser to set the 'stream_file' property.

Constructors

- **SimulatorOptionsParser**
  `public` **SimulatorOptionsParser**`( )`
  - **Description**
    JCommando generated constructor.

Methods

- **createExecuteGrouping**
  `private org.jcommando.Grouping` **createExecuteGrouping**`( )`
  - **Description**
    Generate the grouping for the 'execute' command.
- **doExecute**
  `public abstract void` **doExecute**`( )`
  - **Description**
    Called by parser to perform the 'execute' command.
- **setCli**
  `public abstract void` **setCli**`( )`

- **Description**

  Called by parser to set the 'cli' property.

- **setCluster_file**
  `public abstract void` **setCluster_file**`( java.lang.String` **cluster_file** `)`

  - **Description**

    Called by parser to set the 'cluster_file' property.

  - **Parameters**

    * `cluster_file` – the value to set.

- **setGui_delay**
  `public abstract void` **setGui_delay**`( long` **gui_delay** `)`

  - **Description**

    Called by parser to set the 'gui_delay' property.

  - **Parameters**

    * `gui_delay` – the value to set.

- **setGui**
  `public abstract void` **setGui**`( )`

  - **Description**

    Called by parser to set the 'gui' property.

- **setHelp**
  `public abstract void` **setHelp**`( )`

  - **Description**

    Called by parser to set the 'help' property.

- **setNumber_firings**
  `public abstract void` **setNumber_firings**`( long` **number_firings** `)`

  - **Description**

    Called by parser to set the 'number_firings' property.

  - **Parameters**

    * `number_firings` – the value to set.

- **setOffset_scale**
  `public abstract void` **setOffset_scale**`( java.lang.String` **offset_scale** `)`

  - **Description**

    Called by parser to set the 'offset_scale' property.

  - **Parameters**

    * `offset_scale` – the value to set.

- **setOffset**
`public abstract void` **setOffset**`( java.lang.String` **offset** `)`

  - **Description**
    Called by parser to set the 'offset' property.
  - **Parameters**
    * `offset` – the value to set.

- **setOut_file**
`public abstract void` **setOut_file**`( java.lang.String` **out_file** `)`

  - **Description**
    Called by parser to set the 'out_file' property.
  - **Parameters**
    * `out_file` – the value to set.

- **setRep**
`public abstract void` **setRep**`( long` **rep** `)`

  - **Description**
    Called by parser to set the 'rep' property.
  - **Parameters**
    * `rep` – the value to set.

- **setRuntime_handler**
`public abstract void` **setRuntime_handler**`( java.lang.String` **runtime_handler** `)`

  - **Description**
    Called by parser to set the 'runtime_handler' property.
  - **Parameters**
    * `runtime_handler` – the value to set.

- **setStream_file**
`public abstract void` **setStream_file**`( java.lang.String` **stream_file** `)`

  - **Description**
    Called by parser to set the 'stream_file' property.
  - **Parameters**
    * `stream_file` – the value to set.

Members inherited from class `org.jcommando.JCommandParser`

- `protected void` **addCommand**(`Command` **arg0**)
- `protected void` **addOption**(`Option` **arg0**)
- `private void` **checkOptions**( )
- `private` **classArgArray**
- `private` **className**
- `protected` **commands**
- `protected` **commandsById**
- `private void` **executeCommands**( )
- `private void` **executeSetters**( )
- `String` **getClassName**( )
- `Command` **getCommandById**(`java.lang.String` **arg0**)
- `LinkedHashMap` **getCommands**( )
- `public Option` **getOptionById**(`java.lang.String` **arg0**)
- `String` **getPackageName**( )
- `void` **init**( )
- `private` **numericParseMessages**
- `protected` **optionsById**
- `protected` **optionsByLong**
- `protected` **optionsByShort**
- `private` **packageName**
- `public void` **parse**(`java.lang.String[]` **arg0**)
- `private void` **parseCommand**(`Command` **arg0**, `java.lang.String` **arg1**)
- `private` **parsedCommand**
- `private` **parsedOptions**
- `private boolean` **parseOption**(`Option` **arg0**, `java.lang.String` **arg1**, `java.lang.String` **arg2**)
- `private Object` **parseOptionArgument**(`Option` **arg0**, `java.lang.String` **arg1**)
- `public void` **printUsage**( )
- `void` **setClassName**(`java.lang.String` **arg0**)
- `void` **setPackageName**(`java.lang.String` **arg0**)
- `private Class` **toClassArray**(`java.lang.Class` **arg0**)
- `private String` **toJavaCase**(`java.lang.String` **arg0**)
- `private` **unparsedArguments**

A.5 Package sim.desc

Package Contents *Page*

Classes

A.5.1 Class ClusterDescriptor

The ClusterDescriptor represents the cluster as an undirected graph

Declaration

public class ClusterDescriptor
extends java.lang.Object

Field summary

theCluster

Constructor summary

ClusterDescriptor(String) Construct the cluster descriptor from the given xml file

Method summary

constructClusterGraph(String) Create the graph by transforming a graph into one with PCVertex vertices.

getPCDesc(Vertex) Create a PCDescriptor from tags in the xml (seen as UserDatum attributes)
getTheGraph()

Fields

- private final edu.uci.ics.jung.graph.UndirectedGraph **theCluster**

Constructors

- **ClusterDescriptor**
  `public` **ClusterDescriptor**`( java.lang.String` **graphFile** `)`
  - **Description**
    Construct the cluster descriptor from the given xml file

Methods

- **constructClusterGraph**
  `private edu.uci.ics.jung.graph.UndirectedGraph` **constructClusterGraph**`( java.lang.Strin` **graphFile** `)`
  - **Description**
    Create the graph by transforming a graph into one with PCVertex vertices.
- **getPCDesc**
  `private PCDescriptor` **getPCDesc**`( edu.uci.ics.jung.graph.Vertex v )`
  - **Description**
    Create a PCDescriptor from tags in the xml (seen as UserDatum attributes)
- **getTheGraph**
  `public edu.uci.ics.jung.graph.UndirectedGraph` **getTheGraph**`( )`

A.5.2 Class FilterDescriptor

The FilterDescriptor provides static information about a filter

Declaration

public class FilterDescriptor
extends java.lang.Object

Field summary

peek Number of data elements needed for each firing
pop Number of data elements removed at each firing
push Number of data elements produced at each firing
work Computation needed for each firing

Constructor summary

FilterDescriptor(int, int, int, double) Construct a FilterDescriptor

Method summary

getPeek()
getPop()
getPush()
getWork()

Fields

- private final int **pop**
  - Number of data elements removed at each firing
- private final int **peek**
  - Number of data elements needed for each firing
- private final int **push**
  - Number of data elements produced at each firing
- private final double **work**
  - Computation needed for each firing

Constructors

- **FilterDescriptor**
  `public` **FilterDescriptor**`( int` **push**`, int` **pop**`, int` **peek**`, double` **work** `)`
  - **Description**
    Construct a FilterDescriptor

Methods

- **getPeek**
  `public int` **getPeek**`( )`
- **getPop**
  `public int` **getPop**`( )`
- **getPush**
  `public int` **getPush**`( )`
- **getWork**
  `public double` **getWork**`( )`

A.5.3 Class InputBuffer

The InputBuffer represents the data storage area for a filter

Declaration

public class InputBuffer
extends java.lang.Object

Field summary

elements The BufferElements
lastUsed
maxSize Maximum number of elements in the buffer at any time

Constructor summary

InputBuffer(int) Construct the InputBuffer

Method summary

ensureEnough(int, double) Used on the input to the stream graph to ensure the first filter can always fire
flush() Remove all elements from the buffer
getFilledPercent()
getFreeSpace() Return the number of elements which could be added to this buffer
getLastUsedTime()
getMaxSize()
getMaxTime()
getTimeAt(int) Get the insertion time of the ith oldest element (0-based) in the buffer
instertData(InputBuffer.BufferElement) Inserts an element into the buffer
isFull()
isSorted()
remove(int) Remove the n oldest elements from the buffer
size()
toString()

Fields

- private final int **maxSize**
  - Maximum number of elements in the buffer at any time
- protected java.util.List **elements**
  - The BufferElements
- private double **lastUsed**

Constructors

- **InputBuffer**
  `public` **InputBuffer**`( int` **max** `)`

– **Description**
  Construct the InputBuffer

Methods

- **ensureEnough**
  `public void` **ensureEnough**`( int` **peek**`, double` **time** `)`
  – **Description**
    Used on the input to the stream graph to ensure the first filter can always fire
  – **Parameters**
    * `peek` –
    * `time` –
- **flush**
  `public void` **flush**`( )`
  – **Description**
    Remove all elements from the buffer
- **getFilledPercent**
  `public double` **getFilledPercent**`( )`
- **getFreeSpace**
  `public int` **getFreeSpace**`( )`
  – **Description**
    Return the number of elements which could be added to this buffer
- **getLastUsedTime**
  `public double` **getLastUsedTime**`( )`
- **getMaxSize**
  `public int` **getMaxSize**`( )`
- **getMaxTime**
  `public double` **getMaxTime**`( )`
- **getTimeAt**
  `public double` **getTimeAt**`( int` **i** `)`
  – **Description**
    Get the insertion time of the ith oldest element (0-based) in the buffer
- **instertData**
  `public void` **instertData**`( InputBuffer.BufferElement` **element** `)`
  – **Description**
    Inserts an element into the buffer
  – **Parameters**

* `element` –

- **isFull**
  `public boolean` **isFull**`( )`

- **isSorted**
  `private boolean` **isSorted**`( )`

- **remove**
  `public void` **remove**`( int` **n** `)`

  - **Description**
    Remove the n oldest elements from the buffer
  - **Parameters**
    * **n** – Number of elements to remove

- **size**
  `public int` **size**`( )`

- **toString**
  `public java.lang.String` **toString**`( )`

A.5.4 Class InputBuffer.BufferElement

Each BufferElement has an associated timestamp

Declaration

public static class InputBuffer.BufferElement
extends java.lang.Object
implements java.lang.Comparable

Field summary

time

Constructor summary

InputBuffer.BufferElement(double)

Method summary

compareTo(InputBuffer.BufferElement)
getTime()
toString()

Fields

- private final double **time**

Constructors

- **InputBuffer.BufferElement**
  `public` **InputBuffer.BufferElement**`( double t )`

Methods

- **compareTo**
  `public int` **compareTo**`( InputBuffer.BufferElement o )`

- **getTime**
  `public double` **getTime**`( )`

- **toString**
  `public java.lang.String` **toString**`( )`

A.5.5 Class PCDescriptor

Describes a PC in a cluster

Declaration

public abstract class PCDescriptor
extends java.lang.Object

All known subclasses

RandomProcPC (in A.5.7, page 109), StaticPC (in A.5.8, page 110)

Field summary

stdSpeed Raw speed of the processor

Constructor summary

PCDescriptor(double) Construct the PCDescriptor

Method summary

getBackgroundProcs(double)
getRawSpeed()
getSpeed(double, int)
getSpeed(double, int, double) Get the speed of the PC at the given time.
getSpeed(int) Get the processor speed given it has numFilters processes running
isLoaded() Return true iff this processor is heavily loaded

Fields

- private final double **stdSpeed**
  - Raw speed of the processor

Constructors

- **PCDescriptor**
  `public PCDescriptor( double speed )`
  - **Description**
    Construct the PCDescriptor

Methods

- **getBackgroundProcs**
  `protected abstract int getBackgroundProcs( double time )`
- **getRawSpeed**
  `public double getRawSpeed( )`
- **getSpeed**
  `public double getSpeed( double t, int numFilters )`
- **getSpeed**
  `public double getSpeed( double time, int numFilters, double filterLoad )`
  - **Description**
    Get the speed of the PC at the given time. Currently does not use filterLoad, which would require too much of a fine-grained analysis.
- **getSpeed**
  `public double getSpeed( int numFilters )`
  - **Description**
    Get the processor speed given it has numFilters processes running
- **isLoaded**
  `public boolean isLoaded( )`
  - **Description**
    Return true iff this processor is heavily loaded

A.5.6 Class PipeDescriptor

Describes connections between PCs in a cluster Not currently used.

Declaration

public class PipeDescriptor
extends java.lang.Object

Constructor summary

PipeDescriptor()

Constructors

- **PipeDescriptor**
  `public` **PipeDescriptor**( )

A.5.7 Class RandomProcPC

RandomProcPc objects are processors which go through stages of being loaded and not loaded. Loaded processors tend to have many more processes running at any given time compared to processors which are not loaded.

Declaration

public class RandomProcPC
extends sim.desc.PCDescriptor (in A.5.5, page 107)

Field summary

hoseDone
isHosed Whether the processor is loaded
lastTime
rand
unHosedDone

Constructor summary

RandomProcPC(double) Construct the RandomProcPC

Method summary

getBackgroundProcs(double)
isLoaded()

Fields

- private boolean **isHosed**
  - Whether the processor is loaded
- private double **hoseDone**
- private double **unHosedDone**

- private static final java.util.Random **rand**
- private double **lastTime**

Constructors

- **RandomProcPC**
  `public` **RandomProcPC**`( double` **speed** `)`
  - **Description**
    Construct the RandomProcPC

Methods

- **getBackgroundProcs**
  `protected abstract int` **getBackgroundProcs**`( double` **time** `)`
- **isLoaded**
  `public boolean` **isLoaded**`( )`
  - **Description copied from PCDescriptor (in A.5.5, page 107)**
    Return true iff this processor is heavily loaded

Members inherited from class `sim.desc.PCDescriptor` (in A.5.5, page 107)

- `protected abstract int` **getBackgroundProcs**`( double` **time** `)`
- `public double` **getRawSpeed**`( )`
- `public double` **getSpeed**`( double` **t**, `int` **numFilters** `)`
- `public double` **getSpeed**`( double` **time**, `int` **numFilters**, `double` **filterLoad** `)`
- `public double` **getSpeed**`( int` **numFilters** `)`
- `public boolean` **isLoaded**`( )`
- `private final` **stdSpeed**

A.5.8 Class StaticPC

Declaration

public class StaticPC
extends sim.desc.PCDescriptor (in A.5.5, page 107)

Constructor summary

StaticPC(double)

Method summary

getBackgroundProcs(double)

Constructors

- **StaticPC**
  `public` **StaticPC(** `double` **speed)**

Methods

- **getBackgroundProcs**
  `protected abstract int` **getBackgroundProcs(** `double` **time)**

Members inherited from class `sim.desc.PCDescriptor` (in A.5.5, page 107)

- `protected abstract int` **getBackgroundProcs(** `double` **time)**
- `public double` **getRawSpeed( )**
- `public double` **getSpeed(** `double` **t,** `int` **numFilters)**
- `public double` **getSpeed(** `double` **time,** `int` **numFilters,** `double` **filterLoad)**
- `public double` **getSpeed(** `int` **numFilters)**
- `public boolean` **isLoaded( )**
- `private final` **stdSpeed**

A.5.9 Class StreamItComputationDescriptor

The "highest-level" descriptor, the StreamItComputationDescriptor contains the StreamIt graph, the cluster information, as well as the RuntimeHandler.

Declaration

public class StreamItComputationDescriptor
extends java.lang.Object

Field summary

finalVertex The final vertex in the stream graph
firingsSinceCheckPoint
firstVertex The first vertex in the stream graph
headToTailDist Distance in stream graph from start node to final node
lastCheckpointTime
lastOutputTime
lastRunTime
output The buffer which the final node outputs to
outputsSinceCheckpoint Whether we are using the market runtime handler or not
resourceMap Current resource map
runtimeHandler The runtime handler
streamItGraph The stream graph
theCluster The cluster
waitForCheckpoint If we are currently "flushing" data out to get to a checkpoint

Constructor summary

StreamItComputationDescriptor(String, String, Simulator, double, double, Simulator.rhType) Construct the StreamItComputationDescriptor

Method summary

checkForCheckpoint() Determine whether we should start flushing data in order to reach checkpoint.
constructStreamGraph(String) Constructs the StreamIt graph, noting the first and last filters in the graph
doCheckpoint(double) Actually run the runtime handler and update the PCs with their new filter sets.
finished()
fire(FilterVertex) Fires the given filter
getEndVertex()
getFilterDescriptor(Vertex) Create a FilterDescriptor from user datum tags in the xml file
getFilterLocation(FilterVertex)
getFirstLast(DirectedGraph)
getOutput()
getStartVertex()
getStreamItGraph()
getTheCluster()
nextToRun() Returns the next filter which is able to perform some action(fire, inject).
pushOutput(FilterVertex) Have the given filter push its output.
runFilter(FilterVertex) Run the filter.
totalDataElements()
updatePCs() Notify the PCs to their load from filters
writeSummary(PrintStream, int)

Fields

- private final ClusterDescriptor **theCluster**
  - The cluster
- private final edu.uci.ics.jung.graph.DirectedGraph **streamItGraph**
  - The stream graph
- private final runtime.RuntimeHandler **runtimeHandler**
  - The runtime handler
- private final InputBuffer **output**
  - The buffer which the final node outputs to
- private java.util.Map **resourceMap**

– Current resource map

- private sim.FilterVertex **finalVertex**
  – The final vertex in the stream graph
- private sim.FilterVertex **firstVertex**
  – The first vertex in the stream graph
- private final int **headToTailDist**
  – Distance in stream graph from start node to final node
- private boolean **waitForCheckpoint**
  – If we are currently "flushing" data out to get to a checkpoint
- private double **lastCheckpointTime**
- private double **lastOutputTime**
- private int **outputsSinceCheckpoint**
  – Whether we are using the market runtime handler or not
- private double **lastRunTime**
- private int **firingsSinceCheckPoint**

Constructors

- **StreamItComputationDescriptor**
  `public` **StreamItComputationDescriptor**`( java.lang.String` **streamitGraphFile**`, java.lang.String` **clusterGraphFile**`, sim.Simulator` **sim**`, double` **offset**`, double` **offset_scale**`, sim.Simulator.rhType` **rht** `)`
  – **Description**
  Construct the StreamItComputationDescriptor

Methods

- **checkForCheckpoint**
  `private void` **checkForCheckpoint**`( )`
  – **Description**
  Determine whether we should start flushing data in order to reach checkpoint.
- **constructStreamGraph**
  `private edu.uci.ics.jung.graph.DirectedGraph` **constructStreamGraph**`( java.lang.String` **graphFile** `)`
  – **Description**
  Constructs the StreamIt graph, noting the first and last filters in the graph

- **doCheckpoint**
  `private void` **doCheckpoint**`( double` **time** `)`

  - **Description**

    Actually run the runtime handler and update the PCs with their new filter sets.

- **finished**
  `public void` **finished**`( )`

- **fire**
  `private void` **fire**`( sim.FilterVertex` **firer** `)`

  - **Description**

    Fires the given filter

  - **Parameters**

    * `firer` – Which filter to fire

- **getEndVertex**
  `public sim.FilterVertex` **getEndVertex**`( )`

- **getFilterDescriptor**
  `private static FilterDescriptor` **getFilterDescriptor**`( edu.uci.ics.jung.graph.Vertex` **v** `)`

  - **Description**

    Create a FilterDescriptor from user datum tags in the xml file

- **getFilterLocation**
  `public sim.PCVertex` **getFilterLocation**`( sim.FilterVertex` **fv** `)`

- **getFirstLast**
  `private void` **getFirstLast**`( edu.uci.ics.jung.graph.DirectedGraph` **theGraph** `)`

- **getOutput**
  `public InputBuffer` **getOutput**`( )`

- **getStartVertex**
  `public sim.FilterVertex` **getStartVertex**`( )`

- **getStreamItGraph**
  `public edu.uci.ics.jung.graph.DirectedGraph` **getStreamItGraph**`( )`

- **getTheCluster**
  `public ClusterDescriptor` **getTheCluster**`( )`

- **nextToRun**
  `public sim.FilterVertex` **nextToRun**`( )`

– **Description**

Returns the next filter which is able to perform some action(fire, inject). Resolves ties arbitrarily. Because the initial input buffer is set to always have enough data elements, there should always be a valid FilterVertex to return (the "head" of the graph).

- **pushOutput**
  `private void` **pushOutput**`( sim.FilterVertex` **firer** `)`

  – **Description**

  Have the given filter push its output.

- **runFilter**
  `public void` **runFilter**`( sim.FilterVertex` **firer** `)`

  – **Description**

  Run the filter. The filter will either fire or push output depending on its state.

- **totalDataElements**
  `public int` **totalDataElements**`( )`

- **updatePCs**
  `private void` **updatePCs**`( )`

  – **Description**

  Notify the PCs to their load from filters

- **writeSummary**
  `public void` **writeSummary**`( java.io.PrintStream` **file**`, int` **rep** `)`

A.6 Package sim

Package Contents *Page*

Interfaces

Classes

A.6.1 Interface FilterListener

FilterListener is an interface for classes which need to listen for events from FilterVertex objects.

Declaration

public interface FilterListener

All known subinterfaces

FilterAgent (in A.1.4, page 71), AdaptiveFilterAgent (in A.1.3, page 69)

All classes known to implement interface

FilterAgent (in A.1.4, page 71)

Method summary

notifyDataWait(double) Notify when filter waits for downstream buffers
notifyFire(double) Notify when filter fires
notifyInject(FilterVertex) Notify when filter receives new input

Methods

- **notifyDataWait**
  `void` **notifyDataWait(** `double` **time)**
  - **Description**
    Notify when filter waits for downstream buffers
- **notifyFire**
  `void` **notifyFire(** `double` **time)**
  - **Description**
    Notify when filter fires
- **notifyInject**
  `void` **notifyInject(** `FilterVertex` **firer)**

- **Description**

  Notify when filter receives new input

A.6.2 Interface SimulationListener

A SimulationListener "listens" to StreamIt simulation for various messages

Declaration

public interface SimulationListener

All known subinterfaces

SimulatorGUI (in A.3.1, page 93), Simulator.ConsoleUI (in A.6.6, page 131)

All classes known to implement interface

SimulatorGUI (in A.3.1, page 93), Simulator.ConsoleUI (in A.6.6, page 131)

Method summary

notifyExit() Called when the simulation exits
notifyOutput(long, double) Called when the simulation produces more outputs (the final node pushed data out)
notifyRun() Called when the simulation begins

Methods

- **notifyExit**
  `void` **notifyExit**`( )`

  - **Description**

    Called when the simulation exits

- **notifyOutput**
  `void` **notifyOutput**`( long` **outputs**`, double` **lastOutputTime** `)`

  - **Description**

    Called when the simulation produces more outputs (the final node pushed data out)

- **notifyRun**
  `void` **notifyRun**`( )`

  - **Description**

    Called when the simulation begins

A.6.3 Class FilterVertex

All vertices in the StreamItGraph are of this type. The FilterVertex includes both static (the FilterDescriptor) and dynamic (computation times, input buffer) information about the filter and its computations.

Declaration

public class FilterVertex
extends edu.uci.ics.jung.graph.impl.SimpleDirectedSparseVertex

Field summary

blocked True when waiting to inject elements
buffer Data input buffer to this filter
currentPipe Which output we are ready to send data to (always 0 for pipeline filters)
doneCompTime Time at the end of the last firing of this filter
filterDesc Static information about the filter
lastCheckpointTime Time of last checkpoint
lastPushTime
listeners The Set of FilterListeners listening to this filter
name
nextPushTime Keeps track of when we can push output to downstream buffer next
numPushedSinceFire Number of elements pushed since last firing
waitForRoom True when we're waiting for a downstream buffer to have space

Constructor summary

FilterVertex(FilterDescriptor, String) Construct a FilterVertex

Method summary

addListener(FilterListener)
canPushNextOutput() Return true iff the next output can be successfully injected into the corresponding downstream filter.
downStreamDoneCompTime() Time when "current" downstream filter will finish its computation.
estimateWorkTime(double, PCVertex) Estimate the amount of work this filter must do given it's computing on the given PC
fire(PCVertex) Fire this filter.
getBuffer()
getFilledPercent() How filled the input buffer is
getFilter()
getLastDoneCompTime()
getLastInjectTime()
getMultiplicity() The number of times this filter must fire in order for its downstream neighbors to all fire, based on their pop values

getName()
getOutgoing()
isBlocked() Return true iff this filter is currently waiting to push output into downstream filter(s).
isFinalVertex() Return true iff this is the final vertex in the StreamIt graph.
notifyCheckPoint(double)
notifyDataWait() Tell listeners when we are stuck waiting for downstream buffers to get space.
notifyFire(double)
notifyInject(FilterVertex)
pushOutput(InputBuffer) Push the next output of this filter downstream
timeToRun(boolean, boolean) Returns the time at which this filter can next begin firing or injecting, based on its input buffer and current computation.
timeToRunInternal(boolean, boolean) Returns the time at which this filter can next begin firing or injecting, based on its input buffer and current computation.
toString()

Fields

- private final desc.FilterDescriptor **filterDesc**
  - Static information about the filter
- private final desc.InputBuffer **buffer**
  - Data input buffer to this filter
- private int **currentPipe**
  - Which output we are ready to send data to (always 0 for pipeline filters)
- private double **doneCompTime**
  - Time at the end of the last firing of this filter
- private final java.util.Set **listeners**
  - The Set of FilterListeners listening to this filter
- private boolean **blocked**
  - True when waiting to inject elements
- private int **numPushedSinceFire**
  - Number of elements pushed since last firing
- public double **nextPushTime**
  - Keeps track of when we can push output to downstream buffer next
- private double **lastCheckpointTime**
  - Time of last checkpoint

- private final java.lang.String **name**
- private double **lastPushTime**
- private boolean **waitForRoom**
  - True when we're waiting for a downstream buffer to have space

Constructors

- **FilterVertex**
  `public` **FilterVertex**`( desc.FilterDescriptor` **descriptor**`, java.lang.String` **n** `)`
  - **Description**
    Construct a FilterVertex

Methods

- **addListener**
  `public void` **addListener**`( FilterListener` **l** `)`
- **canPushNextOutput**
  `private boolean` **canPushNextOutput**`( )`
  - **Description**
    Return true iff the next output can be successfully injected into the corresponding downstream filter.
- **downStreamDoneCompTime**
  `private double` **downStreamDoneCompTime**`( )`
  - **Description**
    Time when "current" downstream filter will finish its computation.
- **estimateWorkTime**
  `private double` **estimateWorkTime**`( double` **time**`, PCVertex` **pc** `)`
  - **Description**
    Estimate the amount of work this filter must do given it's computing on the given PC
  - **Parameters**
    * `time` – The time at which the computation begins
    * `pc` – Where the computation is done
  - **Returns** – work time estimate
- **fire**
  `public void` **fire**`( PCVertex` **pc** `)`

- **Description**

  Fire this filter. Removes appropriate number of inputs from the buffer, and injects appropriate number of outputs to its downstream neighbors.

- **getBuffer**
  `public desc.InputBuffer` **getBuffer**`( )`

- **getFilledPercent**
  `public double` **getFilledPercent**`( )`

  - **Description**

    How filled the input buffer is

  - **Returns** –

- **getFilter**
  `public desc.FilterDescriptor` **getFilter**`( )`

- **getLastDoneCompTime**
  `public double` **getLastDoneCompTime**`( )`

- **getLastInjectTime**
  `public double` **getLastInjectTime**`( )`

- **getMultiplicity**
  `public double` **getMultiplicity**`( )`

  - **Description**

    The number of times this filter must fire in order for its downstream neighbors to all fire, based on their pop values

- **getName**
  `public java.lang.String` **getName**`( )`

- **getOutgoing**
  `private java.util.List` **getOutgoing**`( )`

- **isBlocked**
  `public boolean` **isBlocked**`( )`

  - **Description**

    Return true iff this filter is currently waiting to push output into downstream filter(s).

- **isFinalVertex**
  `public boolean` **isFinalVertex**`( )`

  - **Description**

    Return true iff this is the final vertex in the StreamIt graph.

- **notifyCheckPoint**
  `public void` **notifyCheckPoint**`( double` **time** `)`

- **notifyDataWait**
  `private void` **notifyDataWait**`( )`

  - **Description**
    Tell listeners when we are stuck waiting for downstream buffers to get space.

- **notifyFire**
  `private void` **notifyFire**`( double` **t** `)`

- **notifyInject**
  `private void` **notifyInject**`( FilterVertex` **firer** `)`

- **pushOutput**
  `public void` **pushOutput**`( desc.InputBuffer` **output** `)`

  - **Description**
    Push the next output of this filter downstream
  - **Parameters**
    * `output` – The output of the entire stream computation
    * `time` – when the computation finished

- **timeToRun**
  `public double` **timeToRun**`( boolean` **waitForCheckpoint**`, boolean` **runSource** `) throws sim.FilterVertex.SourceException`

  - **Description**
    Returns the time at which this filter can next begin firing or injecting, based on its input buffer and current computation. Returns a negative number if it is unknown when the next time it will fire (possibly because there not enough data elements in its input buffer).

- **timeToRunInternal**
  `public double` **timeToRunInternal**`( boolean` **waitForCheckpoint**`, boolean` **runSource** `) throws sim.FilterVertex.SourceException`

  - **Description**
    Returns the time at which this filter can next begin firing or injecting, based on its input buffer and current computation. Returns a negative number if it is unknown when the next time it will fire (possibly because there not enough data elements in its input buffer).

- **toString**
  `public java.lang.String` **toString**`( )`

Members inherited from class `edu.uci.ics.jung.graph.impl.SimpleDirectedSparseVertex`

- `protected void` **addNeighbor_internal**`( edu.uci.ics.jung.graph.Edge` **arg0**`, edu.uci.ics.jung.graph.Vertex` **arg1** `)`
- `public Edge` **findEdge**`( edu.uci.ics.jung.graph.Vertex` **arg0** `)`

- public Set **findEdgeSet**(edu.uci.ics.jung.graph.Vertex **arg0**)
- protected Collection **getEdges_internal**( )
- public Set **getInEdges**( )
- protected Collection **getNeighbors_internal**( )
- public Set **getOutEdges**( )
- public Set **getPredecessors**( )
- protected Map **getPredsToInEdges**( )
- public Set **getSuccessors**( )
- protected Map **getSuccsToOutEdges**( )
- public int **inDegree**( )
- protected void **initialize**( )
- public boolean **isDest**(edu.uci.ics.jung.graph.Edge **arg0**)
- public boolean **isPredecessorOf**(edu.uci.ics.jung.graph.Vertex **arg0**)
- public boolean **isSource**(edu.uci.ics.jung.graph.Edge **arg0**)
- public boolean **isSuccessorOf**(edu.uci.ics.jung.graph.Vertex **arg0**)
- private **mPredsToInEdges**
- private **mSuccsToOutEdges**
- public int **numPredecessors**( )
- public int **numSuccessors**( )
- public int **outDegree**( )
- protected void **removeNeighbor_internal**(edu.uci.ics.jung.graph.Edge **arg0**, edu.uci.ics.jung.graph.Vertex **arg1**)
- protected void **setPredsToInEdges**(java.util.Map **arg0**)
- protected void **setSuccsToOutEdges**(java.util.Map **arg0**)

Members inherited from class edu.uci.ics.jung.graph.impl.AbstractSparseVertex

- static void ( )
- protected abstract void **addNeighbor_internal**(edu.uci.ics.jung.graph.Edge **arg0**, edu.uci.ics.jung.graph.Vertex **arg1**)
- public ArchetypeVertex **copy**(edu.uci.ics.jung.graph.ArchetypeGraph **arg0**)
- public ArchetypeEdge **findEdge**(edu.uci.ics.jung.graph.ArchetypeVertex **arg0**)
- public Edge **findEdge**(edu.uci.ics.jung.graph.Vertex **arg0**)
- public Set **findEdgeSet**(edu.uci.ics.jung.graph.ArchetypeVertex **arg0**)
- public Set **findEdgeSet**(edu.uci.ics.jung.graph.Vertex **arg0**)
- private static **nextGlobalVertexID**
- protected abstract void **removeNeighbor_internal**(edu.uci.ics.jung.graph.Edge **arg0**, edu.uci.ics.jung.graph.Vertex **arg1**)
- public String **toString**( )

Members inherited from class `edu.uci.ics.jung.graph.impl.AbstractArchetypeVertex`

- `public ArchetypeVertex` **copy**(`edu.uci.ics.jung.graph.ArchetypeGraph` **arg0**)
- `public int` **degree**( )
- `public boolean` **equals**(`java.lang.Object` **arg0**)
- `public ArchetypeEdge` **findEdge**(`edu.uci.ics.jung.graph.ArchetypeVertex` **arg0**)
- `public Set` **findEdgeSet**(`edu.uci.ics.jung.graph.ArchetypeVertex` **arg0**)
- `protected abstract Collection` **getEdges_internal**( )
- `public ArchetypeVertex` **getEqualVertex**(`edu.uci.ics.jung.graph.ArchetypeGraph` **arg0**)
- `public ArchetypeVertex` **getEquivalentVertex**(`edu.uci.ics.jung.graph.ArchetypeGraph` **arg0**)
- `public Set` **getIncidentEdges**( )
- `public Set` **getIncidentElements**( )
- `protected abstract Collection` **getNeighbors_internal**( )
- `public Set` **getNeighbors**( )
- `public boolean` **isIncident**(`edu.uci.ics.jung.graph.ArchetypeEdge` **arg0**)
- `public boolean` **isNeighborOf**(`edu.uci.ics.jung.graph.ArchetypeVertex` **arg0**)
- `public int` **numNeighbors**( )

Members inherited from class `edu.uci.ics.jung.graph.impl.AbstractElement`

- `protected void` **addGraph_internal**(`AbstractArchetypeGraph` **arg0**)
- `void` **checkIDs**(`java.util.Map` **arg0**)
- `public ArchetypeGraph` **getGraph**( )
- `int` **getID**( )
- `public int` **hashCode**( )
- `protected` **id**
- `protected void` **initialize**( )
- `protected` **m_Graph**
- `protected void` **removeGraph_internal**( )

Members inherited from class `edu.uci.ics.jung.utils.UserDataDelegate`

- `static void` ( )
- `public void` **addUserDatum**(`java.lang.Object` **arg0**, `java.lang.Object` **arg1**, `UserDataContainer.CopyAction` **arg2**)
- `public Object` **clone**( ) `throws java.lang.CloneNotSupportedException`
- `public boolean` **containsUserDatumKey**(`java.lang.Object` **arg0**)
- `protected static` **factory**
- `public Object` **getUserDatum**(`java.lang.Object` **arg0**)

- public UserDataContainer.CopyAction **getUserDatumCopyAction**(java.lang.Object **arg0**)
- public Iterator **getUserDatumKeyIterator**( )
- public void **importUserData**(UserDataContainer **arg0**)
- public Object **removeUserDatum**(java.lang.Object **arg0**)
- public static void **setUserDataFactory**(UserDataFactory **arg0**)
- public void **setUserDatum**(java.lang.Object **arg0**, java.lang.Object **arg1**, UserDataContainer.CopyAction **arg2**)
- protected **udc_delegate**

A.6.4 Class PCVertex

The PCVertex represents a vertex in the cluster graph. It uses the set of filters currently on it to determine effective speed of its associated PC.

Declaration

public class PCVertex
extends edu.uci.ics.jung.graph.impl.SimpleUndirectedSparseVertex

Field summary

filters The filters running on this resource
myNum
name
num
pcDesc The actual PCDescriptor info

Constructor summary

PCVertex(PCDescriptor, String) Create a PCVertex

Method summary

getDesriptor()
getFilterLoad() Get the total load on this resource from the filters
getNum()
getNumFilters() Number of filters currently on this resource
getSpeed(double) Get the speed of this resource at the given time.
setFilters(Set)
toString()

Fields

- private final desc.PCDescriptor **pcDesc**
  - The actual PCDescriptor info
- private java.util.Set **filters**

- The filters running on this resource

- private static int **num**
- private final int **myNum**
- private final java.lang.String **name**

Constructors

- **PCVertex**
  `public` **PCVertex**`( desc.PCDescriptor` **descriptor**`, java.lang.String` **n** `)`
  - **Description**
    Create a PCVertex

Methods

- **getDesriptor**
  `public desc.PCDescriptor` **getDesriptor**`( )`
- **getFilterLoad**
  `public double` **getFilterLoad**`( )`
  - **Description**
    Get the total load on this resource from the filters
  - **Returns** – amount of total work
- **getNum**
  `public int` **getNum**`( )`
- **getNumFilters**
  `public int` **getNumFilters**`( )`
  - **Description**
    Number of filters currently on this resource
- **getSpeed**
  `public double` **getSpeed**`( double` **time** `)`
  - **Description**
    Get the speed of this resource at the given time.
  - **Parameters**
    * `time` – Current time
  - **Returns** –
- **setFilters**
  `public void` **setFilters**`( java.util.Set` **newFilters** `)`
- **toString**
  `public java.lang.String` **toString**`( )`

Members inherited from class edu.uci.ics.jung.graph.impl.SimpleUndirectedSparseVertex

- protected void **addNeighbor_internal**(edu.uci.ics.jung.graph.Edge **arg0**, edu.uci.ics.jung.graph.Vertex **arg1**)
- public Edge **findEdge**(edu.uci.ics.jung.graph.Vertex **arg0**)
- public Set **findEdgeSet**(edu.uci.ics.jung.graph.Vertex **arg0**)
- protected Collection **getEdges_internal**( )
- public Set **getInEdges**( )
- protected Collection **getNeighbors_internal**( )
- protected Map **getNeighborsToEdges**( )
- public Set **getOutEdges**( )
- public Set **getPredecessors**( )
- public Set **getSuccessors**( )
- public int **inDegree**( )
- protected void **initialize**( )
- public boolean **isDest**(edu.uci.ics.jung.graph.Edge **arg0**)
- public boolean **isPredecessorOf**(edu.uci.ics.jung.graph.Vertex **arg0**)
- public boolean **isSource**(edu.uci.ics.jung.graph.Edge **arg0**)
- public boolean **isSuccessorOf**(edu.uci.ics.jung.graph.Vertex **arg0**)
- private **mNeighborsToEdges**
- public int **numPredecessors**( )
- public int **numSuccessors**( )
- public int **outDegree**( )
- protected void **removeNeighbor_internal**(edu.uci.ics.jung.graph.Edge **arg0**, edu.uci.ics.jung.graph.Vertex **arg1**)
- protected void **setNeighborsToEdges**(java.util.Map **arg0**)

Members inherited from class edu.uci.ics.jung.graph.impl.AbstractSparseVertex

- static void ( )
- protected abstract void **addNeighbor_internal**(edu.uci.ics.jung.graph.Edge **arg0**, edu.uci.ics.jung.graph.Vertex **arg1**)
- public ArchetypeVertex **copy**(edu.uci.ics.jung.graph.ArchetypeGraph **arg0**)
- public ArchetypeEdge **findEdge**(edu.uci.ics.jung.graph.ArchetypeVertex **arg0**)
- public Edge **findEdge**(edu.uci.ics.jung.graph.Vertex **arg0**)
- public Set **findEdgeSet**(edu.uci.ics.jung.graph.ArchetypeVertex **arg0**)
- public Set **findEdgeSet**(edu.uci.ics.jung.graph.Vertex **arg0**)
- private static **nextGlobalVertexID**
- protected abstract void **removeNeighbor_internal**(edu.uci.ics.jung.graph.Edge **arg0**, edu.uci.ics.jung.graph.Vertex **arg1**)
- public String **toString**( )

Members inherited from class edu.uci.ics.jung.graph.impl.AbstractArchetypeVertex

- public ArchetypeVertex **copy**(edu.uci.ics.jung.graph.ArchetypeGraph **arg0**)
- public int **degree**( )
- public boolean **equals**(java.lang.Object **arg0**)
- public ArchetypeEdge **findEdge**(edu.uci.ics.jung.graph.ArchetypeVertex **arg0**)
- public Set **findEdgeSet**(edu.uci.ics.jung.graph.ArchetypeVertex **arg0**)
- protected abstract Collection **getEdges_internal**( )
- public ArchetypeVertex **getEqualVertex**(edu.uci.ics.jung.graph.ArchetypeGraph **arg0**)
- public ArchetypeVertex **getEquivalentVertex**(edu.uci.ics.jung.graph.ArchetypeGraph **arg0**)
- public Set **getIncidentEdges**( )
- public Set **getIncidentElements**( )
- protected abstract Collection **getNeighbors_internal**( )
- public Set **getNeighbors**( )
- public boolean **isIncident**(edu.uci.ics.jung.graph.ArchetypeEdge **arg0**)
- public boolean **isNeighborOf**(edu.uci.ics.jung.graph.ArchetypeVertex **arg0**)
- public int **numNeighbors**( )

Members inherited from class edu.uci.ics.jung.graph.impl.AbstractElement

- protected void **addGraph_internal**(AbstractArchetypeGraph **arg0**)
- void **checkIDs**(java.util.Map **arg0**)
- public ArchetypeGraph **getGraph**( )
- int **getID**( )
- public int **hashCode**( )
- protected **id**
- protected void **initialize**( )
- protected **m_Graph**
- protected void **removeGraph_internal**( )

Members inherited from class edu.uci.ics.jung.utils.UserDataDelegate

- static void ( )
- public void **addUserDatum**(java.lang.Object **arg0**, java.lang.Object **arg1**, UserDataContainer.CopyAction **arg2**)
- public Object **clone**( ) throws java.lang.CloneNotSupportedException
- public boolean **containsUserDatumKey**(java.lang.Object **arg0**)
- protected static **factory**
- public Object **getUserDatum**(java.lang.Object **arg0**)

- public UserDataContainer.CopyAction **getUserDatumCopyAction**(java.lang.Object **arg0**)
- public Iterator **getUserDatumKeyIterator**( )
- public void **importUserData**(UserDataContainer **arg0**)
- public Object **removeUserDatum**(java.lang.Object **arg0**)
- public static void **setUserDataFactory**(UserDataFactory **arg0**)
- public void **setUserDatum**(java.lang.Object **arg0**, java.lang.Object **arg1**, UserDataContainer.CopyAction **arg2**)
- protected **udc_delegate**

A.6.5 Class Simulator

The main class and entry point of the StreamIt simulation

Declaration

public class Simulator
extends java.lang.Object

Field summary

computation The computation
lastOutputTime Last time we incremented outputs
listeners The listeners
options Command-line options
outputs Total outputs

Constructor summary

Simulator(Simulator.SimulatorOptions) Constructs a Simulator

Method summary

addListener(SimulationListener)
getLastOutputTime()
getOutputs()
getStreamIt()
getStreamItGraph()
main(String[]) Creates a Simulator object.
notifyFire()
notifyOutput()
simulate(long) Simulates the StreamIt computation, notifying listeners of pertinent events.
sleepSafe(long)
writeFooter(PrintStream) Write footer information to the octave script
writeSummary(PrintStream, int) Write summary information to the octave script

Fields

- private desc.StreamItComputationDescriptor **computation**
  - The computation
- private java.util.Set **listeners**
  - The listeners
- private final Simulator.SimulatorOptions **options**
  - Command-line options
- private long **outputs**
  - Total outputs
- private double **lastOutputTime**
  - Last time we incremented outputs

Constructors

- **Simulator**
  `public` **Simulator**`( Simulator.SimulatorOptions` **opts** `)`
  - **Description**
    Constructs a Simulator

Methods

- **addListener**
  `private void` **addListener**`( SimulationListener` **listener** `)`
- **getLastOutputTime**
  `public double` **getLastOutputTime**`( )`
- **getOutputs**
  `public long` **getOutputs**`( )`
- **getStreamIt**
  `public desc.StreamItComputationDescriptor` **getStreamIt**`( )`
- **getStreamItGraph**
  `public edu.uci.ics.jung.graph.Graph` **getStreamItGraph**`( )`
- **main**
  `public static void` **main**`( java.lang.String[]` **args** `)`
  - **Description**
    Creates a Simulator object. Run the simulation the given number of times.
- **notifyFire**
  `private void` **notifyFire**`( )`

- **notifyOutput**
  `private void` **notifyOutput**`( )`
- **simulate**
  `public void` **simulate**`( long` **rep** `)`
  - **Description**
    Simulates the StreamIt computation, notifying listeners of pertinent events.
- **sleepSafe**
  `private void` **sleepSafe**`( long` **time** `)`
- **writeFooter**
  `private static void` **writeFooter**`( java.io.PrintStream` **file** `)`
  - **Description**
    Write footer information to the octave script
- **writeSummary**
  `private void` **writeSummary**`( java.io.PrintStream` **file**`, int` **rep** `)`
  - **Description**
    Write summary information to the octave script

A.6.6 Class Simulator.ConsoleUI

ConsoleUI displays throughput information to the console

Declaration

private static class Simulator.ConsoleUI
extends java.lang.Object
implements SimulationListener

Field summary

numOut

Constructor summary

Simulator.ConsoleUI()

Method summary

notifyExit()
notifyOutput(long, double)
notifyRun()

Fields

- private int **numOut**

Constructors

- **Simulator.ConsoleUI**
  `private` **Simulator.ConsoleUI**`( )`

Methods

- **notifyExit**
  `void` **notifyExit**`( )`
  - **Description copied from SimulationListener (in A.6.2, page 117)**
    Called when the simulation exits

- **notifyOutput**
  `void` **notifyOutput**`(` `long` **outputs**`,` `double` **lastOutputTime** `)`
  - **Description copied from SimulationListener (in A.6.2, page 117)**
    Called when the simulation produces more outputs (the final node pushed data out)

- **notifyRun**
  `void` **notifyRun**`( )`
  - **Description copied from SimulationListener (in A.6.2, page 117)**
    Called when the simulation begins

A.6.7 Class Simulator.rhType

Declaration

public static final class Simulator.rhType
extends java.lang.Enum

Field summary

dynamicRH
marketRH
staticRH

Constructor summary

Simulator.rhType()

Method summary

valueOf(String)
values()

Fields

- public static final Simulator.rhType **marketRH**
- public static final Simulator.rhType **staticRH**
- public static final Simulator.rhType **dynamicRH**

Constructors

- **Simulator.rhType**
  `private` **Simulator.rhType**`( )`

Methods

- **valueOf**
  `public static Simulator.rhType` **valueOf**`( java.lang.String` **name** `)`
- **values**
  `public static final Simulator.rhType[]` **values**`( )`

Members inherited from class `java.lang.Enum`

- `protected final Object` **clone**`( ) throws CloneNotSupportedException`
- `public final int` **compareTo**`( Enum` **arg0** `)`
- `public final boolean` **equals**`( Object` **arg0** `)`
- `public final Class` **getDeclaringClass**`( )`
- `public final int` **hashCode**`( )`
- `private final` **name**
- `public final String` **name**`( )`
- `private final` **ordinal**
- `public final int` **ordinal**`( )`
- `public String` **toString**`( )`
- `public static Enum` **valueOf**`( Class` **arg0**`, String` **arg1** `)`

A.6.8 Class Simulator.SimulatorOptions

Handles command line options and defaults

Declaration

public static class Simulator.SimulatorOptions
extends sim.parse.SimulatorOptionsParser (in A.4.1, page 96)

Field summary

clusterFile
dataOut
doCLI
doGUI
guiDelay
numFirings
offset
offset_scale
outFile
reps
rh
streamFile
validArgs

Constructor summary

Simulator.SimulatorOptions()

Method summary

comment(String, Object[])
doCLI()
doExecute()
doGUI()
experimentName()
getClusterFile()
getDelay()
getFile()
getNumFirings()
getOffset()
getOffsetScale()
getReps()
getRuntimeHandler()
getStreamFile()
setCli()
setCluster_file(String)
setGui_delay(long)
setGui()
setHelp()
setNumber_firings(long)
setOffset_scale(String)
setOffset(String)
setOut_file(String)
setRep(long)
setRuntime_handler(String)
setStream_file(String)
validArgs()

writeHeaderInfo()

Fields

- private java.io.PrintStream **dataOut**
- private java.lang.String **streamFile**
- private java.lang.String **clusterFile**
- private java.lang.String **rh**
- private long **numFirings**
- private long **reps**
- private long **guiDelay**
- private boolean **doCLI**
- private boolean **doGUI**
- private java.lang.String **outFile**
- private double **offset**
- private double **offset_scale**
- private boolean **validArgs**

Constructors

- **Simulator.SimulatorOptions**
  `public` **Simulator.SimulatorOptions**`( )`

Methods

- **comment**
  `private void` **comment**`( java.lang.String` **com**`, java.lang.Object[]` **args** `)`

- **doCLI**
  `public boolean` **doCLI**`( )`
- **doExecute**
  `public abstract void` **doExecute**`( )`
  - **Description copied from parse.SimulatorOptionsParser (in A.4.1, page 96)**
    Called by parser to perform the 'execute' command.
- **doGUI**
  `public boolean` **doGUI**`( )`

- **experimentName**
  `private java.lang.String` **experimentName**`( )`

- **getClusterFile**
  `public java.lang.String` **getClusterFile**`( )`

- **getDelay**
  `public long` **getDelay**`( )`

- **getFile**
  `public java.io.PrintStream` **getFile**`( )`

- **getNumFirings**
  `public long` **getNumFirings**`( )`

- **getOffset**
  `public double` **getOffset**`( )`

- **getOffsetScale**
  `public double` **getOffsetScale**`( )`

- **getReps**
  `public long` **getReps**`( )`

- **getRuntimeHandler**
  `public java.lang.String` **getRuntimeHandler**`( )`

- **getStreamFile**
  `public java.lang.String` **getStreamFile**`( )`

- **setCli**
  `public abstract void` **setCli**`( )`

  - **Description copied from parse.SimulatorOptionsParser** (**in A.4.1, page 96**)

    Called by parser to set the 'cli' property.

- **setCluster_file**
  `public abstract void` **setCluster_file**`( java.lang.String` **cluster_file** `)`

  - **Description copied from parse.SimulatorOptionsParser** (**in A.4.1, page 96**)

    Called by parser to set the 'cluster_file' property.

  - **Parameters**

    * `cluster_file` – the value to set.

- **setGui_delay**
  `public abstract void` **setGui_delay**`( long` **gui_delay** `)`

  - **Description copied from parse.SimulatorOptionsParser** (**in A.4.1, page 96**)

    Called by parser to set the 'gui_delay' property.

- **Parameters**
  - gui_delay – the value to set.

- **setGui**
  `public abstract void setGui( )`
  - **Description copied from parse.SimulatorOptionsParser** (**in A.4.1, page 96**)
    Called by parser to set the 'gui' property.

- **setHelp**
  `public abstract void setHelp( )`
  - **Description copied from parse.SimulatorOptionsParser** (**in A.4.1, page 96**)
    Called by parser to set the 'help' property.

- **setNumber_firings**
  `public abstract void setNumber_firings( long number_firings )`
  - **Description copied from parse.SimulatorOptionsParser** (**in A.4.1, page 96**)
    Called by parser to set the 'number_firings' property.
  - **Parameters**
    - number_firings – the value to set.

- **setOffset_scale**
  `public abstract void setOffset_scale( java.lang.String offset_scale )`
  - **Description copied from parse.SimulatorOptionsParser** (**in A.4.1, page 96**)
    Called by parser to set the 'offset_scale' property.
  - **Parameters**
    - offset_scale – the value to set.

- **setOffset**
  `public abstract void setOffset( java.lang.String offset )`
  - **Description copied from parse.SimulatorOptionsParser** (**in A.4.1, page 96**)
    Called by parser to set the 'offset' property.
  - **Parameters**
    - offset – the value to set.

- **setOut_file**
  `public abstract void setOut_file( java.lang.String out_file )`

- **Description copied from parse.SimulatorOptionsParser (in A.4.1, page 96)**

  Called by parser to set the 'out_file' property.
- **Parameters**
  * `out_file` – the value to set.

- **setRep**
  `public abstract void` **setRep**`( long` **rep** `)`
  - **Description copied from parse.SimulatorOptionsParser (in A.4.1, page 96)**

    Called by parser to set the 'rep' property.
  - **Parameters**
    * `rep` – the value to set.

- **setRuntime_handler**
  `public abstract void` **setRuntime_handler**`( java.lang.String` **runtime_handler** `)`
  - **Description copied from parse.SimulatorOptionsParser (in A.4.1, page 96)**

    Called by parser to set the 'runtime_handler' property.
  - **Parameters**
    * `runtime_handler` – the value to set.

- **setStream_file**
  `public abstract void` **setStream_file**`( java.lang.String` **stream_file** `)`
  - **Description copied from parse.SimulatorOptionsParser (in A.4.1, page 96)**

    Called by parser to set the 'stream_file' property.
  - **Parameters**
    * `stream_file` – the value to set.

- **validArgs**
  `public boolean` **validArgs**`( )`

- **writeHeaderInfo**
  `private void` **writeHeaderInfo**`( )`

Members inherited from class `sim.parse.SimulatorOptionsParser` (in A.4.1, page 96)

- `private Grouping` **createExecuteGrouping**`( )`
- `public abstract void` **doExecute**`( )`
- `public abstract void` **setCli**`( )`
- `public abstract void` **setCluster_file**`( java.lang.String` **cluster_file** `)`

- public abstract void **setGui_delay**(long **gui_delay**)
- public abstract void **setGui**( )
- public abstract void **setHelp**( )
- public abstract void **setNumber_firings**(long **number_firings**)
- public abstract void **setOffset_scale**(java.lang.String **offset_scale**)
- public abstract void **setOffset**(java.lang.String **offset**)
- public abstract void **setOut_file**(java.lang.String **out_file**)
- public abstract void **setRep**(long **rep**)
- public abstract void **setRuntime_handler**(java.lang.String **runtime_handler**)
- public abstract void **setStream_file**(java.lang.String **stream_file**)

Members inherited from class org.jcommando.JCommandParser

- protected void **addCommand**(Command **arg0**)
- protected void **addOption**(Option **arg0**)
- private void **checkOptions**( )
- private **classArgArray**
- private **className**
- protected **commands**
- protected **commandsById**
- private void **executeCommands**( )
- private void **executeSetters**( )
- String **getClassName**( )
- Command **getCommandById**(java.lang.String **arg0**)
- LinkedHashMap **getCommands**( )
- public Option **getOptionById**(java.lang.String **arg0**)
- String **getPackageName**( )
- void **init**( )
- private **numericParseMessages**
- protected **optionsById**
- protected **optionsByLong**
- protected **optionsByShort**
- private **packageName**
- public void **parse**(java.lang.String[] **arg0**)
- private void **parseCommand**(Command **arg0**, java.lang.String **arg1**)
- private **parsedCommand**
- private **parsedOptions**
- private boolean **parseOption**(Option **arg0**, java.lang.String **arg1**, java.lang.String **arg2**)
- private Object **parseOptionArgument**(Option **arg0**, java.lang.String **arg1**)
- public void **printUsage**( )
- void **setClassName**(java.lang.String **arg0**)
- void **setPackageName**(java.lang.String **arg0**)
- private Class **toClassArray**(java.lang.Class **arg0**)
- private String **toJavaCase**(java.lang.String **arg0**)
- private **unparsedArguments**

A.6.9 Class Util

The Util class provides static utility methods. It cannot be instantiated.

Declaration

public class Util
extends java.lang.Object

Constructor summary

Util()

Method summary

allMappings(Set, Set) Returns a set containing all possible mappings from keyset to valueSet which have exactly 1 mapping per element of keySet
getDataDouble(UserDataContainer, String) Gets a double in a UserDataContainer, given a key
getDataInt(UserDataContainer, String) Gets an int in a UserDataContainer, given a key
getDistance(DirectedGraph, SimpleDirectedSparseVertex, SimpleDirectedSparseVertex, int)
getGraphDist(DirectedGraph, SimpleDirectedSparseVertex, SimpleDirectedSparseVertex) Distance from start node to final node along the longest path
pow(double, int) Wrapper around Math.pow
reverseMapLookup(Map, V) Given a map and a value, returns all keys in the map which are mapped to this value.
Sum(Collection) Returns the sum of a collection of Doubles

Constructors

- **Util**
  `private` **Util**`( )`

Methods

- **allMappings**
  `public static java.util.Set` **allMappings**`( java.util.Set` **keySet**`, java.util.Set` **valueSet** `)`
  - **Description**
    Returns a set containing all possible mappings from keyset to valueSet which have exactly 1 mapping per element of keySet
  - **Parameters**
    * `keySet` – the set of keys
    * `valueSet` – the set of values

- **Returns** – a Set of all possible mappings whose keysets are all identical to the given keyset and whose value collection is a subset of the given valueSet

- **getDataDouble**
  `public static double` **getDataDouble**`( edu.uci.ics.jung.utils.UserDataContainer v, java.lang.String` **key** `)`

  - **Description**
    Gets a double in a UserDataContainer, given a key
  - **Parameters**
    * `v` – Data container
    * `key` – The key

- **getDataInt**
  `public static int` **getDataInt**`( edu.uci.ics.jung.utils.UserDataContainer v, java.lang.String` **key** `)`

  - **Description**
    Gets an int in a UserDataContainer, given a key
  - **Parameters**
    * `v` – Data container
    * `key` – The key

- **getDistance**
  `private static int` **getDistance**`( edu.uci.ics.jung.graph.DirectedGraph` **graph**`, edu.uci.ics.jung.graph.impl.SimpleDirectedSparseVertex` **cur**`, edu.uci.ics.jung.graph.imp` **end**`, int` **distance** `)`

- **getGraphDist**
  `public static int` **getGraphDist**`( edu.uci.ics.jung.graph.DirectedGraph` **graph**`, edu.uci.ics.jung.graph.impl.SimpleDirectedSparseVertex` **start**`, edu.uci.ics.jung.graph.in` **end** `)`

  - **Description**
    Distance from start node to final node along the longest path

- **pow**
  `private static double` **pow**`( double` **b**`, int` **e** `)`

  - **Description**
    Wrapper around Math.pow
  - **Parameters**
    * `b` – base
    * `e` – exponent
  - **Returns** – b∧e

- **reverseMapLookup**
  `public static java.util.Set` **reverseMapLookup**`( java.util.Map` **map**`, java.lang.Object` **value** `)`
  - **Description**
    Given a map and a value, returns all keys in the map which are mapped to this value.
  - **Parameters**
    * `map` – The map
    * `value` – The value
- **Sum**
  `public static double` **Sum**`( java.util.Collection` **numbers** `)`
  - **Description**
    Returns the sum of a collection of Doubles

A.6.10 Exception FilterVertex.SourceException

Declaration

public static class FilterVertex.SourceException
extends java.lang.Exception

Constructor summary

FilterVertex.SourceException()

Constructors

- **FilterVertex.SourceException**
  `public` **FilterVertex.SourceException**`( )`

Members inherited from class `java.lang.Exception`

- `static final` **serialVersionUID**

Members inherited from class `java.lang.Throwable`

- `private transient` **backtrace**
- `private` **cause**
- `private` **detailMessage**
- `public synchronized native Throwable` **fillInStackTrace**`( )`
- `public Throwable` **getCause**`( )`
- `public String` **getLocalizedMessage**`( )`
- `public String` **getMessage**`( )`
- `private synchronized StackTraceElement` **getOurStackTrace**`( )`
- `public StackTraceElement` **getStackTrace**`( )`

- private native int **getStackTraceDepth**( )
- private native StackTraceElement **getStackTraceElement**(int **arg0**)
- public synchronized Throwable **initCause**(Throwable **arg0**)
- public void **printStackTrace**( )
- public void **printStackTrace**(java.io.PrintStream **arg0**)
- public void **printStackTrace**(java.io.PrintWriter **arg0**)
- private void **printStackTraceAsCause**(java.io.PrintStream **arg0**, StackTraceElement[] **arg1**)
- private void **printStackTraceAsCause**(java.io.PrintWriter **arg0**, StackTraceElement[] **arg1**)
- private static final **serialVersionUID**
- public void **setStackTrace**(StackTraceElement[] **arg0**)
- private **stackTrace**
- public String **toString**( )
- private synchronized void **writeObject**(java.io.ObjectOutputStream **arg0**) throws java.io.IOException

Bibliography

[1] Josh Aas. Understanding the linux 2.6.8.1 cpu scheduler. `http://citeseer.ist.psu.edu/aas05understanding.html`.

[2] A. AuYoung, B. Chun, A. Snoeren, and A. Vahdat. Resource allocation in federated distributed computing infrastructures. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the Ondemand IT InfraStructure*, October 2004.

[3] Magdalena Balazinska, Hari Balakrishnan, and Mike Stonebraker. Contract-Based Load Management in Federated Distributed Systems. In *1st Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, March 2004.

[4] Eric B. Baum and Igor Durdanovic. Evolution of cooperative problem solving in an artificial economy. *Neural Computation*, 12(12):2743–2775, 2001.

[5] Jiawen Chen, Michael I. Gordon, William Thies, Matthias Zwicker, Kari Pulli, and Frédo Durand. A reconfigurable architecture for load-balanced rendering. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 71–80, New York, NY, USA, 2005. ACM Press.

[6] Trishul M. Chilimbi and Martin Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 199–209, New York, NY, USA, 2002. ACM Press.

[7] B. Chun. *Market-Based Cluster Resource Management.* PhD thesis, University of California at Berkeley, 2001.

[8] Brent N. Chun, Philip Buonadonna, Alvin AuYoung, Chaki Ng, David C. Parkes, Jeffrey Shneidman, Alex C. Snoeren, and Amin Vahdat. Mirage: A Microeconomic Resource Allocation System for Sensornet Testbeds. In *Proceedings of 2nd IEEE Workshop on Embedded Networked Sensors (EmNetsII)*, 2005.

[9] Donald Ferguson, Yechiam Yemini, and Christos Nikolaou. Microeconomic Algorithms for Load Balancing in Distributed Computer Systems. In *International Conference on Distributed Computer Systems*, pages 491–499, 1988.

[10] Michael Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Christopher Leger, Andrew A. Lamb, Jeremy Wong, Henry Hoffman, David Z. Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA USA, October 2002.

[11] Bernardo A Huberman and Tad Hogg. Distributed computation as an economic system. *Journal of Economic Perspectives*, 9(1):141–52, Winter 1995. available at `http://ideas.repec.org/a/aea/jecper/v9y1995i1p141-52.html`.

[12] Jung. Jung: Java universal network/graph framework. `http://jung.sourceforge.net/`.

[13] Michal Karczmarek, William Thies, and Saman Amarasinghe. Phased scheduling of stream programs. In *Languages, Compilers, and Tools for Embedded Systems*, San Diego, CA, June 2003.

[14] Bernardo A. Huberman Kevin Lai and Leslie Fine. Tycoon: A Distributed Market-based Resource Allocation System. Technical Report arXiv:cs.DC/0404013, HP Labs, Palo Alto, CA, USA, April 2004.

[15] Blake LeBaron. Agent-based computational finance: Suggested readings and early research. *Journal of Economic Dynamics and Control*, 24(5-7):679–702,

June 2000. available at http://ideas.repec.org/a/eee/dyncon/v24y2000i5-7p679-702.html.

[16] Blake LeBaron. Calibrating an agent-based financial market. `http://people.brandeis.edu/~blebaron/wps.html`, 04 2002.

[17] Roger Lewin. *Complexity. Life at the Edge of Chaos.* Macmillan Publishing Company, New York, 1992.

[18] Janis Sermulins, William Thies, Rodric Rabbah, and Saman Amarasinghe. Cache aware optimization of stream programs. In *LCTES'05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 115–126, New York, NY, USA, 2005. ACM Press.

[19] Jeffrey Shneidman, Chaki Ng, David C. Parkes, Alvin AuYoung, Alex C. Snoeren, Amin Vahdat, and Brent Chun. Why markets could (but don't currently) solve resource allocation problems in systems. In *Proceedings of Tenth Workshop on Hot Topics in Operating Systems) [HotOS-X 2005]*, June 2005.

[20] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction*, Grenoble, France, April 2002.

[21] Carl A. Waldspurger, Tad Hogg, Bernardo A. Huberman, Jeffrey O. Kephart, and W. Scott Stornetta. Spawn: A distributed computational economy. *Software Engineering*, 18(2):103–117, 1992.