Scalable Audio Feature Extraction

Devon Leno Bryant

A Thesis Presented to the Faculty of
Univeristy of Colorado at Colorado Springs
in Candidacy for the Degree of
Master of Science in Computer Science

Adviser: Dr. Rory Lewis
April 2014

Recommended for Acceptance by

The Department of Computer Science

Dr. Rory Lewis

Date

Dr. Jugal Kalita

Date

Dr. Terrance Boult

Date

Abstract

Feature extraction provides the foundation for many systems and research projects dealing with machine learning. Extracting features from audio data can be computationally expensive and time consuming, particularly with larger sets of audio data.

This paper describes *Safe*, a system for scalable and efficient audio feature extraction in the Music Information Retrieval domain. The system builds upon previous research in efficient audio feature extraction, taking advantage of both multicore processing and distributed cluster computing to provide a scalable architecture for feature extraction algorithms.

Lastly, the system's performance is compared with current leading extraction libraries and a theoretical model of the system's scalability is developed.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The *Music Information Retrieval* (MIR) domain spans a wide variety of disciplines and applications. MIR techniques are being leveraged by businesses and academics for recommendation systems, classification, identification, automatic transcription, similarity searching, and more. Despite the seemingly different nature of these problems, many of these systems actually leverage similar audio feature extraction algorithms at their core. These feature extraction algorithms can range from low-level digital signal processing techniques to high-level machine learning algorithms. Most of these algorithms are computationally intensive, both in resource usage and running times.

The amount of audio data and music being produced today is significant. Companies such as Spotify are adding over 20,000 songs per day to their existing catalog of over 20 million songs [1]. The current Apple iTunes catalog has over 26 million songs [2]. Most of the existing MIR libraries and algorithms are not easily scalable to work with larger datasets such as these.

Numerous libraries exist for audio feature extraction and many have focused on algorithmic optimizations for efficient extraction, often using C++ for its performance benefits in numerical computing. However, as modern computing continues to shift

[1] http://press.spotify.com/us/information/

[2] http://www.apple.com/pr/library/2013/02/06iTunes-Store-Sets-New-Record-with-25-Billion-Songs-Sold.html

towards more parallel architectures [2], it is clear that different programming models and techniques are needed to take full advantage of these systems. Often, the imperative techniques used for optimal execution on a single processor are ill-suited or difficult to transition to parallel environments [24].

Additional research has been done in extending existing libraries to work in distributed computing environments [20, 10, 4]. These systems are all based on a master/worker division of nodes, which suffers from two main drawbacks. The primary drawback is that a master node in any system provides a single point of failure, meaning these systems do not address issues with fault-tolerance. The second drawback is that a master node in a system often enforces bounds on the scalability; introducing an upper bound when throughput is limited by the master node and forcing a lower bound of two nodes from the master/worker separation.

This paper describes a high-performance concurrent system for audio feature extraction based on modern techniques in parallel computing and reactive programming. Scalability of the system is fault-tolerant and addresses both vertical scalability through multicore processing and horizontal scalability through distributed clustering. While the primary focus of this research is on performance and scalability, there is a strong secondary goal towards providing a general extensible MIR framework for others to re-use and build upon.

Chapter 2

Background and Related Work

There have been many MIR feature extraction libraries developed over the last decade, each with different feature sets and focus. This section outlines some of the most popular and relevant libraries in audio feature extraction, though many more exist.

One of the earliest libraries to provide audio feature extraction that is still used today is Marsyas [25]. Marsyas is an efficient C++ framework for general dataflow audio processing that has been used for various analysis and synthesis tasks. Marsyas provides both a command-line interface and a graphical editor for building data flows. Marsyas has also been extended to perform batch extraction in distributed environments [4]. The distributed model relies on a custom dispatcher node sending audio clips over TCP to known worker nodes for feature extraction.

Yaafe [19] is a fast and efficient library for audio feature extraction written mostly in C++ with a Python interface. The focus of Yaafe is on efficiency, utilizing dataflow graphs to eliminate redundant calculations between different features. Yaafe is mostly a command line tool and supports output in CSV or HDF5 formats.

jAudio [22] is a Java-based audio extraction framework developed primarily for MIR researchers. The framework provides a GUI for selecting features and filling out parameters. Results can be exported to XML or Weka's ARFF [1] format for further

[1] http://www.cs.waikato.ac.nz/ml/weka/arff.html

experimentation or machine learning. jAudio also provides a mechanism for describing dependencies between features, similar to Yaafe, for the purpose of eliminating duplicate calculations.

OMEN [20] is a distributed extraction framework that leverages jAudio as its base extraction engine. The distribution model for OMEN is based on a hierarchy of different node types: a master node provides the overall coordination, library nodes store collections of audio data, and worker nodes perform the actual feature extraction. All communication in OMEN is handled through SOAP-based web services.

Sonic Annotator [6] is a command-line tool for running batch feature extraction on large numbers of audio files. Sonic Annotator is based on VAMP [2] plugins, a general framework and C++ API for audio feature extraction algorithms. Numerous VAMP plugin libraries have been developed by different research groups and are available for use. Sonic Visualiser, a complimentary tool to Sonic Annotator, provides a helpful user interface for feature exploration and many different graphical displays for visualizing features. Unfortunately, algorithms and results in VAMP cannot be shared between plugins which leads to some inefficiencies during extraction.

iSoundMW [10] is a distributed feature extraction framework and similarity search engine. The distribution model uses a single master node that sends small segmented frames of audio data to registered client worker nodes for feature extraction.

MIRtoolbox [18] is a Matlab-based MIR feature extraction library that takes advantage of existing visualizations and algorithms available in the Matlab ecosystem. MIRtoolbox provides some explicit mechanisms for composing features in stages from lower-level or intermediary calculations. However, memory management is an issue since the entire file and all feature calculations are held in memory. MIRtoolbox provides some parallel extraction capabilities, allowing concurrent extraction of separate files.

[2] http://www.vamp-plugins.org/

openSMILE [12] is a C++ based library focused on real-time Speech and MIR feature extraction. Batch extraction capabilities are provided as well as multi-threaded support for parallel extraction on a single machine. openSMILE is based on a complex shared data memory system which allows single-component writes and multi-component reads of matrix data and leverages ring-buffers to manage memory. Different output formats are provided including CSV, ARFF, LibSVM [3], and HTK [4].

[3] http://www.csie.ntu.edu.tw/ cjlin/libsvm/

[4] http://www.ee.columbia.edu/ln/LabROSA/doc/HTKBook21/HTKBook.html

Chapter 3

System Description

3.1 Feature Representations

The primary abstraction for any extraction library is the description of a *Feature*. Within the Safe system, each feature is described by a set of required and optional parameters needed to perform its calculations. Features in audio processing are often built on top of other features or share common steps in their calculations. For example, the *Fast Fourier Transform* (FFT) is generally a shared step for any algorithms working in the frequency domain. For that reason, each feature also describes its own dataflow as a sequence of smaller computational steps. This dataflow representation provides two major benefits. The first benefit is algorithm reuse; new algorithms can leverage existing code for common calculations. The second benefit is that duplicate calculations between features can often be eliminated through simple dataflow analysis on the provided feature sets. Section 4.1 describes this dataflow analysis technique in more detail. For a full list of implemented features, parameter descriptions, and dataflow sequences see Appendix A.

Externally, features are described through a simple syntax similar to the syntax used by Yaafe [19]. Listing 1 shows a simplified Backus-Naur Form (BNF) of the feature extraction syntax used by the system and listing 2 shows an example entry of this syntax. Each entry contains three primary sections:

1. *name* - A unique name for the feature extraction.
2. *feature* - The feature to be extracted and any feature parameters such as the step size, window type, etc.
3. *output* - Optional output parameters such as the directory to write to or the precision of the results.

Listing 1 Simplified Feature Syntax BNF

```
        ⟨identifier⟩  ⊨  { A...Z | a...z | 0...9 }
              ⟨name⟩  ⊨  ⟨identifier⟩
           ⟨feature⟩  ⊨  ⟨identifier⟩
         ⟨parameter⟩  ⊨  ⟨identifier⟩ ”=” ( ⟨quoted string⟩ | ⟨number⟩ )
        ⟨parameters⟩  ⊨  ⟨parameter⟩ { ”,” ⟨parameter⟩ }
⟨feature definition⟩  ⊨  ⟨name⟩ ”:” ⟨feature⟩ [ ⟨parameters⟩ ] [ ”-out” ⟨parameters⟩ ]
```

Listing 2 MFCC Feature Extraction Description

```
mfcc: MFCC frameSize=256, stepSize=256, windowType=”hamming”
```

3.2 System Interfaces

Within the system, there are two primary interfaces for running feature extraction over existing datasets. The first interface provides command-line execution for running bulk extraction on a single machine. Listing 3 outlines the syntax and parameters used in the command-line interface.

The two required parameters for any extraction are:

Listing 3 Command Line Syntax

```
-i <file> | --input <file>
      input path (file or directory) for audio file(s) to process
-r | --recursive
      flag to process audio data in sub-directories (default = false)
-p <file> | --plan <file>
      path to feature extraction plan file
-f <value> | --feature <value>
      description and parameters for a single feature to extract
-s <value> | --sample-rate <value>
      sample rate in Hz of all audio files (default = 44100)
-o <file> | --output-dir <file>
      directory to write featre output to (default = './')
-m | --metrics
      flag to capture/print metrics (default = false)
```

1. *Input* - The individual audio file or directory containing audio files to run extraction on.

2. *Plan File* or *Feature* - A file containing the feature set and parameters or an inline description of the feature and its parameters.

The remaining command-line parameters are optional and use the specified defaults if no input is provided. One particular parameter of interest is the *sample rate*. Safe currently requires all audio data to be encoded with the same sample rate. This limitation exists because several features use the expected sample rate to pre-compute values for use in their calculations. For example, the *Mel-Frequency Cepstral Coefficients* (MFCC) algorithm uses the sample rate along with other parameters to pre-compute and cache its mel filter bank. In cases where a dataset contains audio with different sample rates, the audio data needs to be pre-processed and re-sampled into a common rate.

The Safe system also has the ability to capture and print out metrics for a given extraction run. If enabled, Safe will capture the following metrics for every sub-feature computation:

- *Total Messages* - The total number of times the sub-feature calculation was called during extraction.

- *Total Time* - The accumulated total time spent calculating the sub-feature.

- *Min, Max, Median, Mean Times* - The minimum, maximum, median, and mean times spent in a single calculation of the sub-feature.

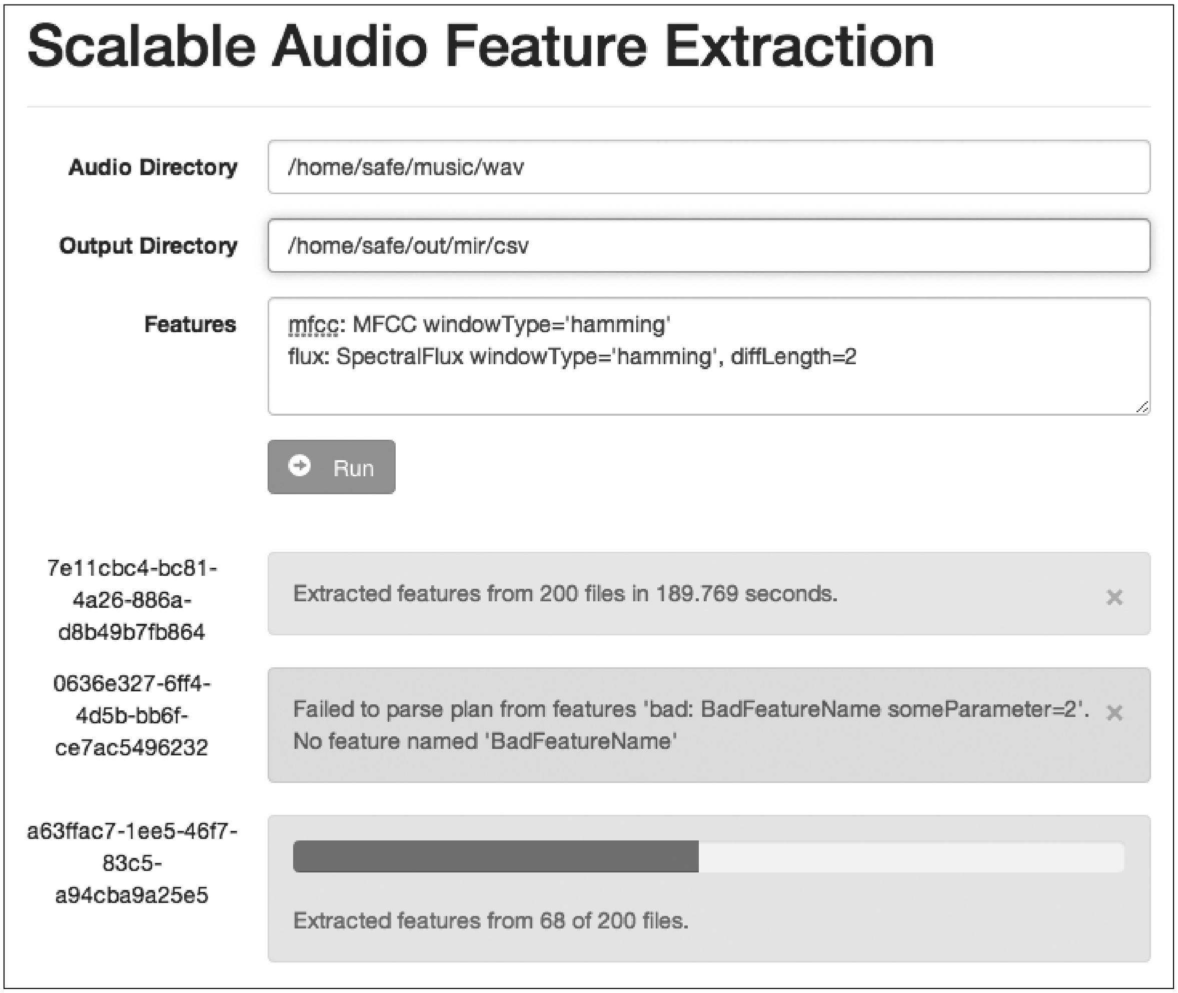
The second system interface, used primarily for controlling extraction in distributed environments, is a web-based user interface. The web interface uses REST [13] communication for executing extraction and monitoring the progress and system status. Each extraction request is assigned a unique id that can be used for querying the status of the extraction. There are three possible responses for extraction status queries:

1. *Ongoing* - Specifies the number of audio files that have been processed along with the total number of files.

2. *Failed* - Gives a description of any failures that occurred during extraction.

3. *Finished* - Specifies the total number of files processed and the time taken to run the extraction (in milliseconds).

Figure 3.2 below shows a screenshot of the web interface for the system. This example screenshot shows the three different status results for extraction requests.



Scalable Audio Feature Extraction

Audio Directory: /home/safe/music/wav

Output Directory: /home/safe/out/mir/csv

Features:

```
mfcc: MFCC windowType='hamming'
flux: SpectralFlux windowType='hamming', diffLength=2
```

7e11cbc4-bc81-4a26-886a-d8b49b7fb864: Extracted features from 200 files in 189.769 seconds.

0636e327-6ff4-4d5b-bb6f-ce7ac5496232: Failed to parse plan from features 'bad: BadFeatureName someParameter=2'. No feature named 'BadFeatureName'

a63ffac7-1ee5-46f7-83c5-a94cba9a25e5: Extracted features from 68 of 200 files.

Figure 3.1: Safe Web Interface

Chapter 4

Efficient and Scalable Extraction

The primary goal of this research is to provide an efficient framework for audio feature extraction that can easily scale in both vertical and horizontal directions. This is achieved by extending the techniques and algorithms used in other MIR libraries and adapting them to work with functional, asynchronous and reactive programming models. Safe is implemented in Scala [1] and leverages the Akka [2] library for parallel and distributed computing.

4.1 Efficient Feature Extraction

As mentioned in chapter 3, features can often be described by dataflow sequences of smaller computational steps or sub-features. Figure 4.1 shows an example of the step sequences for calculating three different features: *Mel-Frequency Cepstral Coefficients* (MFCC) [7], *Constant-Q Transform* (CQT) [5], and *Spectral Shape* [14].

By breaking down features into smaller steps, efficiencies can be gained by eliminating shared calculations. A set of multiple features in the system is combined to form a *Feature Plan*. When shared calculations between feature sequences are

[1] http://www.scala-lang.org/

[2] http://akka.io/

eliminated in a feature extraction plan, the plan naturally forms a directed acyclic dataflow graph. This is a common technique in dataflow analysis that is used by other MIR feature extraction libraries [21, 19] to eliminate redundant calculations. This technique is similar to the problem of Common Subexpression Elimination [3] in compiler optimization. Figure 4.2 below shows an example of the optimized dataflow extraction plan for the same three features described earlier.

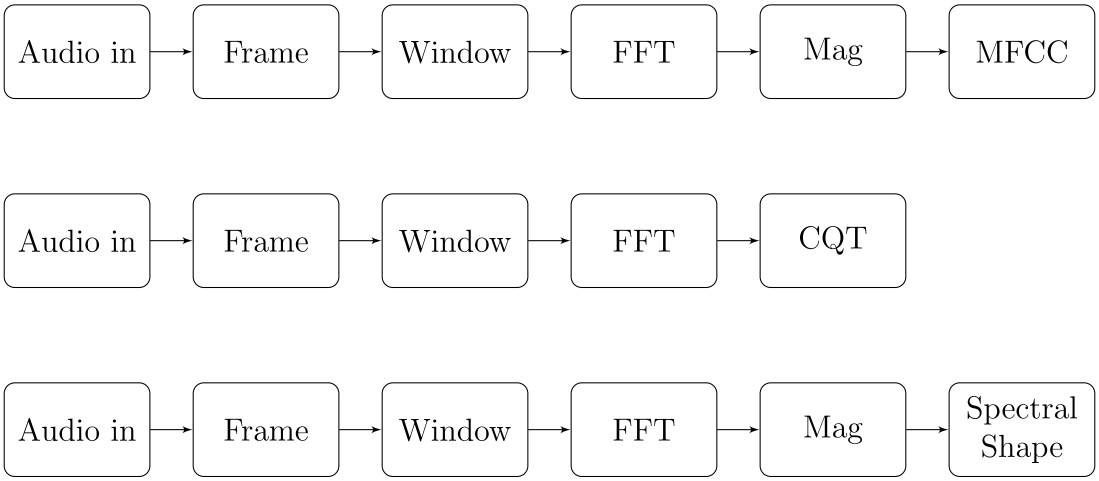


Figure 4.1: Sequential dataflow for MFCC, CQT, and Spectral Shape features

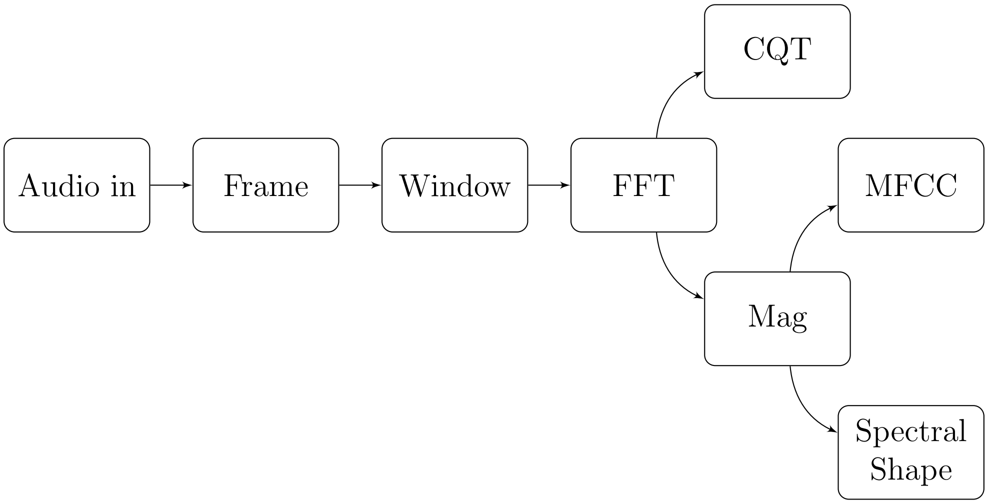


Figure 4.2: Optimal dataflow extraction plan

[3] http://en.wikipedia.org/wiki/Common_subexpression_elimination

Feature extraction plans in Safe are encoded using *adjacency lists*. Each plan holds a single feature or sub-feature computation and zero or more downstream plans to pass results to. Algorithm 1 shows the basic data representation for feature extraction plans. Algorithm 2 shows the heuristic algorithm that is used to combine multiple features or plans into optimal dataflow graphs.

Algorithm 1 Feature Extraction Plan Representation

```
case class Plan(feature: Feature, children: Seq[Plan])
```

Algorithm 2 Dataflow Graph Optimization

```
def dataflow(plans: Seq[Plan]): Seq[Plan] =
  plans.foldLeft(Nil)((merged, plan) => merge(merged, Seq(plan)))

def merge(as: Seq[Plan], bs: Seq[Plan]): Seq[Plan] = {
  val merged = for {
    a <- as
    b <- bs if (a.feature == b.feature)
  } yield Plan(a.feature, merge(a.children, b.children))

  (as ++ bs).filterNot(
      p => merged.exists(_.feature == p.feature)) ++ merged
}
```

4.2 Parallel Feature Extraction

With the processing model expressed as a directed acyclic dataflow graph, audio data can be streamed through the pipeline to extract the various features. This approach lends itself to numerous levels of parallelism:

- *File Level* - Extraction can be run concurrently on multiple files.
- *Plan Level* - Any time a plan branches into separate paths, each path can run concurrently on the results of the parent calculation.
- *Feature Level* - When audio data is split into separate chunks (frames), each stage of a feature extraction sequence can concurrently operate on a different section of the data.

File level parallelism is a fairly self-explanatory concept and is the approach used in [18]. It is the most obvious form of parallelism and requires little to no coordination between processes. Tables 4.1 and 4.2 show examples of plan level and feature level parallelism where each row represents a different processor and each column represents possible computations over time. With plan level parallelism, different steps can run concurrent calculations on the same data outputs of a parent step. For example, table 4.1 shows that the CQT and Magnitude calculations can run concurrently on the output of the FFT. With feature level parallelism, a given calculation may be executing on a section of audio data concurrently with downstream calculations running off previous results. In table 4.2, the example highlights that a Window function can pass its output to the FFT function and immediately begin working on the next frame. *Note*: These tables are used for illustrative purposes of where parallelism can occur and do not represent the actual thread scheduling and division within the system. In the actual running system, some steps will take much longer than others and a single thread may finish multiple steps in the time another thread takes to execute one. Additionally, different combinations of file, plan, and feature level concurrency may occur together.

FFT($window_1$)	Magnitude(fft_1)	MFCC(mag_1)
	CQT(fft_1)	Spectral Shape(mag_1)

Table 4.1: Plan level parallelism

$frame_1$	Window($frame_1$)	FFT($window_1$)	CQT(fft_1)
	$frame_2$	Window($frame_2$)	FFT($window_2$)
		$frame_3$	Window($frame_3$)
			$frame_4$

Table 4.2: Feature level parallelism

Managing these different levels of concurrency with traditional thread scheduling and lock management would be complicated and difficult. Instead, an Actor model [1] for concurrency is used.

Actors are objects that encapsulate state and behavior and communicate exclusively through messaging. Messaging between actors is never direct. Instead, communication occurs through the use of recipient mailboxes. Each actor has exactly one mailbox and messages are dequeued and processed by the actor in a serial fashion. This serial processing guarantee helps eliminate synchronization and locking concerns within the actor's actual processing logic. Conceptually, actors can be thought of as having their own light-weight isolated threads. It is the responsibility of the actor container to manage the actual scheduling and execution against real threads.

Actors also provide a rich hierarchy and supervision system. Each actor can create any number of child actors to execute sub-tasks. When an actor creates another actor, it becomes a supervisor and is responsible for delegation and control of the child actor. When a child actor fails for any reason, the supervisor (parent) is notified of the failure and is responsible for handling the failure. When a subordinate fails, the supervising actor has the option to resume the subordinate (keeping any existing state), restart the subordinate (clearing out any previous state), terminate the subordinate, or escalate the failure (failing itself).

Actor models, being asynchronous and message-based, provide a natural encoding for stream based dataflow processing. Within Safe, each computational step in the dataflow graph is represented by an actor and the intermediary results (messages) between actors are immutable. By encoding the feature extraction dataflow graphs as actor hierarchies in this fashion, all three levels of parallelism mentioned (file, plan, and feature) are provided with no additional components required. The dynamic nature of Akka's actor system make it simple to create complex fault-tolerant feature extraction hierarchies. Within a feature extraction plan, each actor usually takes on one of four common messaging roles described in [17]:

- *Splitter* - Actors that take a single message or result and split the work into smaller pieces for downstream processing. Splitters provide much of the parallelism in feature extraction. File level parallelism is provided by splitting a message with multiple file locations (or a directory) into individual messages for each file. Feature level parallelism is provided by a Frame actor, splitting an individual file into smaller chunks (frames) to perform further computations on.

- *Transform* - The most common actor type in a feature extraction plan is a transform actor. These actors provide simple mapping functions from one type to another and send the result on for further processing. Examples of transforms are Windowing functions, FFT, Power Spectrum, MFCC, Spectral Shape, etc.

- *Resequencer* - Actors that take a stream of potentially unordered messages and send them out in an ordered sequence. Sequencing is important when writing out frame-based extracted features. For example, when writing out to a CSV file the features need to be in the same order as the frames in the source audio file.

- *Aggregator* - Aggregators are used to calculate features that go across multiple frames in an audio file or across an entire dataset. Some example aggregate functions are Spectral Flux or any max/min/average/total/etc. calculation across audio files.

By default, a single actor is created for each node in the dataflow graph and the actor messaging hierarchy mirrors the arcs in the graph. For many systems, the concurrency provided by the file, plan, and feature level divisions is sufficient. However, for highly concurrent systems (systems with a large number of processors), the Splitter and Transform actors can be replaced with actor pools and a round-robin routing strategy. This strategy allows concurrent execution of the same computational step on different frames. For example, multiple FFT calculations could be run concurrently on different frames. In the current system, Resequencers and Aggregators cannot be placed into pools because they need to maintain a temporary state across multiple messages. Because of this requirement, scalability is often limited by features that require aggregate calculations.

4.3 Distributed Feature Extraction

The distribution model for the system leverages Akka's clustering framework [4]. Akka's cluster system provides a fault-tolerant decentralized membership service based on Amazon's Dynamo [9] store. The cluster architecture uses gossip protocols [26] to manage node membership within the cluster. Using the gossip protocol, the current state of the cluster is gossiped between nodes and preference is given to members that have not yet seen the latest version. Initiation into a cluster is handled

[4] http://doc.akka.io/docs/akka/2.2.1/common/cluster.html

by a node sending a join message to one of the existing nodes in the cluster. In order to provide a fault-tolerant solution, failures in the cluster need to be detected and communicated to the rest of the system. For this, Akka uses a Phi Accrual Failure Detector [16].

While previous research in distributed audio feature extraction exists [20, 10, 4], they are all based on a master/worker division of nodes. By leveraging a decentralized and masterless clustering framework, there is no single point of failure or single point bottlenecks in the system. Since the clustering framework is Actor based as well, it is simple to scale the feature extraction framework from a single machine to multiple nodes.

4.3.1 Localized Data

The simplest form of distributed feature extraction assumes that a collection of audio files is already distributed across the cluster onto the local file systems. This configuration is similar to the library node concept used by OMEN [20]. From there, bulk extraction can be performed across the cluster through a simple broadcast message to all nodes in the cluster. While this provides a simple and scalable solution, this approach is not fault-tolerant or flexible. This distribution model couples the extraction execution to the machines containing the data. If a node is down or fails, there is no way to run extraction over the audio data stored on that node.

In this case, the time taken to extract all features will be limited by the longest running node in the cluster. For optimal extraction times, the data needs to be evenly balanced across the cluster and nodes within the cluster should be homogeneous, i.e. each machine should have an equal number of computing resources.

4.3.2 Distributed Data

A more robust and fault-tolerant model for distributed extraction requires a distributed file system such as Hadoop HDFS [23]. Hadoop provides a whole suite of capabilities for distributed analysis and transformation of data using the MapReduce [8] paradigm. However, Safe's cluster extraction only leverages the distributed file system from Hadoop.

In HDFS, there are two types of nodes: *NameNodes* and *DataNodes*. *NameNodes* provide the overall management and coordination of directory hierarchies and location of the data within HDFS. File content is split into chunks, typically 128 mb in size, and those chunks are stored on the local file systems of *DataNodes*. Fault-tolerance in HDFS is provided through replication. Individual chunks are replicated and distributed onto multiple *DataNodes*. In a standard configuration *NameNodes* in a hadoop cluster provide a single point of failure, though a High Availability (HA) feature is provided that allows redundant *NameNodes* to be configured for fast failover. HDFS clients typically access files through a TCP protocol, but a short-circuiting mechanism [5] is provided for cases when the data and processing are co-located on the same machine.

Figure 4.3.2 shows the distributed deployment architecture of the system using HDFS. Since the Safe cluster is masterless, any number of nodes in the cluster can be configured to serve http requests. Safe nodes can be co-located with HDFS nodes or run on separate clusters.

When an extraction request is made to one of the nodes in the Safe cluster, the receiving node will query the *NameNode* for all the paths of the audio data and route extraction of individual audio files to different nodes in the cluster. The primary routing logic uses a Round-Robin routing strategy to distribute the extraction work-

[5] http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/ShortCircuitLocalReads.html

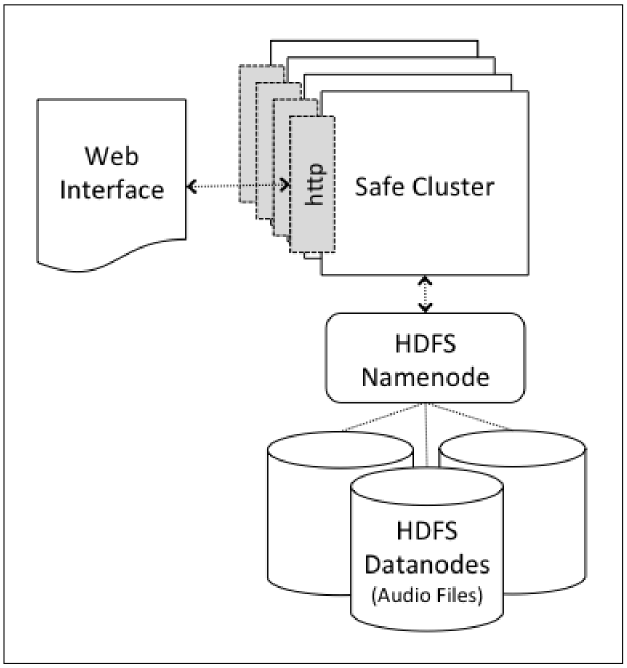


Figure 4.3: Safe Distributed Deployment

load across the cluster. Some experimentation with routing strategies that take data locality into account was also performed. In this case, the routing logic favors sending extraction requests to the same machine where the data is located, taking advantage of the HDFS short-circuit reads. When different nodes in a cluster are unbalanced, i.e. some nodes have a larger heap, more CPU, etc., an adaptive load balancing router should be used instead to distribute feature extraction. Akka provides a probabilistic router based on weighting from the following metrics:

- *Heap* - Remaining JVM heap capacity.
- *Load* - System load average over the past minute.
- *CPU* - Percentage of CPU utilization.

Chapter 5

Evaluation and Results

5.1 System Evaluation

In order to evaluate the system's performance and scalability, a large dataset of music audio was required. The dataset used for testing was a private collection of 1,332 songs, approximately 5,120 minutes of audio. All files were encoded in CD-quality 44.1 KHz 16-bit stereo WAV format. The overall dataset was split into smaller subsets to allow for testing on varying sized datasets. Table 5.1 shows the statistics of all the subsets used in testing.

Audio Files	Total (minutes)	Max (minutes)	Min (minutes)	Avg (minutes)
1	2.5	2.5	2.5	2.5
2	5.0	2.5	2.5	2.5
4	10	3.49	1.51	2.5
9	20	4.04	1.09	2.22
17	40	4.56	0.92	2.35
35	80	4.56	0.59	2.29
61	160	5.91	0.59	2.62
107	320	6.98	0.41	2.99
200	640	11.35	0.41	3.2
371	1280	11.35	0.41	3.45
707	2560	15.72	0.23	3.62
1332	5120	32.25	0.11	3.84

Table 5.1: Evaluation Dataset

Safe was compared with three other feature extraction libraries for performance: Yaafe [1], openSMILE [2], and Sonic Annotator [3]. Yaafe and openSMILE in particular were chosen because of their strong performance characteristics. Table 5.2 shows the versions of each library used for this evaluation. Initial evaluations also included MIRToolbox [4], though its performance was significantly worse and the results have been left out of these benchmarks.

Library	Version
Yaafe	0.64
openSMILE	2.0-rc1
Sonic Annotator	1.0
Safe	0.1

Table 5.2: Extraction Libraries and Versions

The benchmark used for comparing the different libraries was a measure of the total time taken to extract six different features and output the results to CSV files. The six features picked for the benchmark were:

- Mel-Frequency Cepstral Coefficients (13 coefficients, 40 filters)
- Spectral Flux
- Spectral Centroid
- Spectral Spread
- Spectral Skewness
- Spectral Kurtosis

This particular feature set was chosen because each library has support for these features and the algorithms are fairly standard and not likely to vary between implementations. For all features, the frame size was 1024 samples and the step (hop) size

[1] http://yaafe.sourceforge.net/

[2] http://opensmile.sourceforge.net/

[3] http://www.vamp-plugins.org/sonic-annotator/

[4] http://www.jyu.fi/hum/laitokset/musiikki/en/research/coe/materials/mirtoolbox

was 512. All of the scripts and configuration used for this evaluation are available online [5]. In the interest of fairness, attempts were made to provide configurations for optimal performance with each library.

Yaafe was compiled with FFTW [6] for fast FFT calculations. Since Yaafe performs dataflow analysis to optimize execution, no other special configuration was required. Since openSMILE has support for multi-threaded concurrency, experimentation was performed to find an optimal number of threads for the test. Ultimately, openSMILE was configured to run with a single thread because all tests with multi-threaded configurations appeared to degrade the performance. For openSMILE, the dataflow graph is encoded explicitly into the configuration since there is no dataflow analysis support. For Sonic Annotator, the LibXtract [7] plugins were used to calculate the spectral shape features (centroid, spread, skewness, and curtosis), while the BBC plugins [8] and Queen Mary University plugins [9] respectively were used to calculate the Spectral Flux and MFCC features. Sonic Annotator does not provide any support for dataflow optimizations, so many of the algorithms are running redundant calculations. For this benchmark Safe was configured using Akka's default Fork-Join executor, which calculates its thread pool size based on the number of processors p and a parallelism factor ν (default 3.0):

$$poolsize = \lceil p\nu \rceil \tag{5.1}$$

All evaluations were run on a Macbook Pro, OS X 10.7.5, with a 2.5 GHz Intel Core i7 processor 8 GB 1333 MHz DDR3 RAM. Figure 5.1 shows the overall time taken by each library to run extraction over the full dataset. Figure 5.2 shows a plot of each libraries performance as the number of songs increases. Yaafe, openSMILE, and

[5] http://github.com/devonbryant/mir-eval-scripts

[6] http://www.fftw.org/

[7] http://github.com/jamiebullock/LibXtract

[8] http://github.com/bbcrd/bbc-vamp-plugins

[9] http://isophonics.net/QMVampPlugins

Safe (single-threaded) are all very close in terms of performance. In this benchmark, the multi-threaded version of Safe is roughly four times faster than the other solutions.

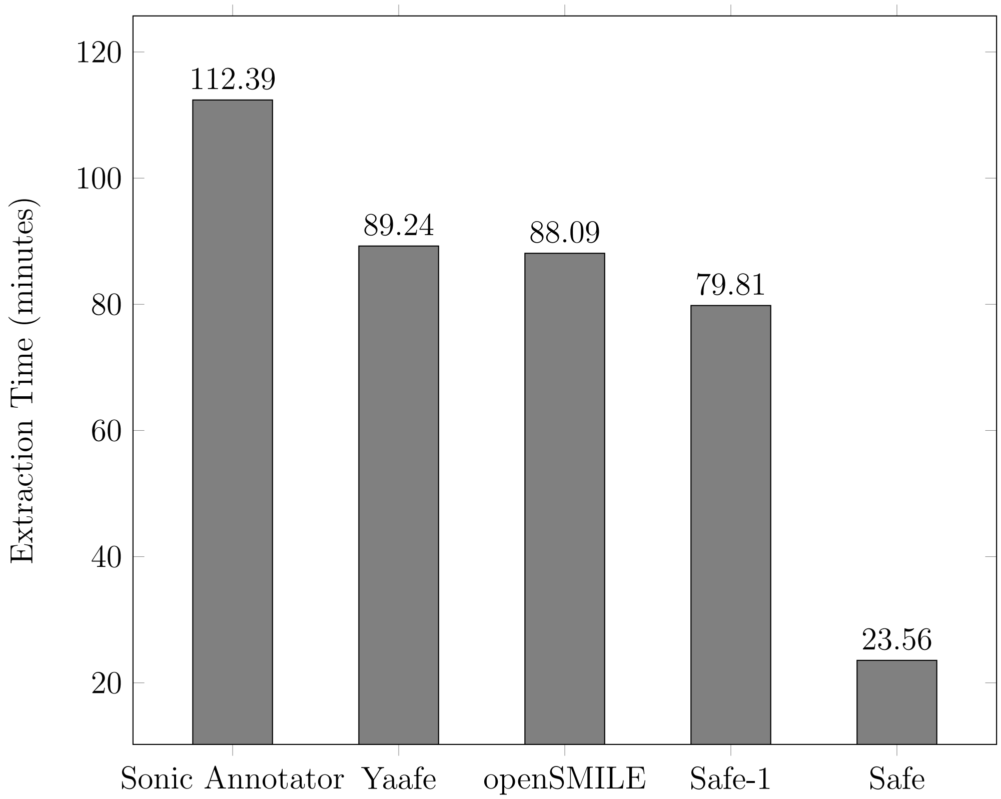


Figure 5.1: Feature extraction times (1332 songs, 5120 minutes of audio)

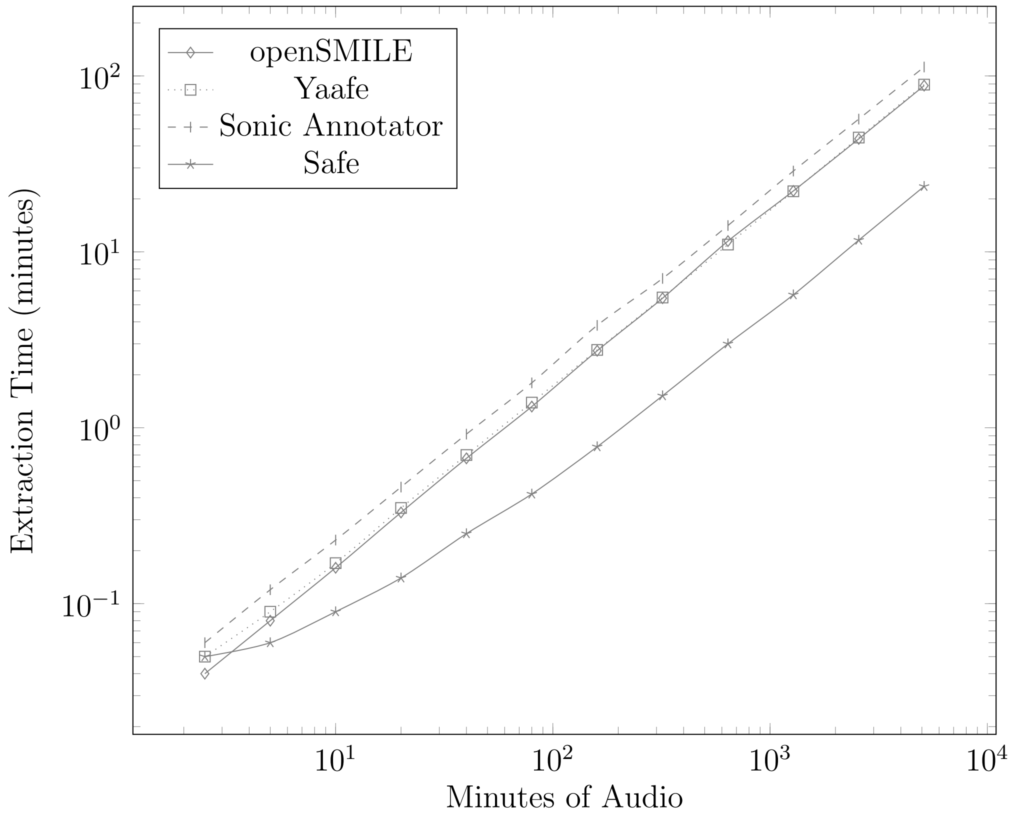


Figure 5.2: Feature extraction times for increasing number of songs

In order to verify that the parallelism in Safe is not limited to concurrent extraction of multiple files, a benchmark was setup to measure extraction times on single audio files of increasing length. Figure 5.3 shows the results of this benchmark. For this benchmark, Safe was configured to use Akka's fixed thread pool dispatcher with five threads. These results indicate that the Safe system has a more significant fixed startup cost than the other libraries but better variable performance with increasing recording lengths.

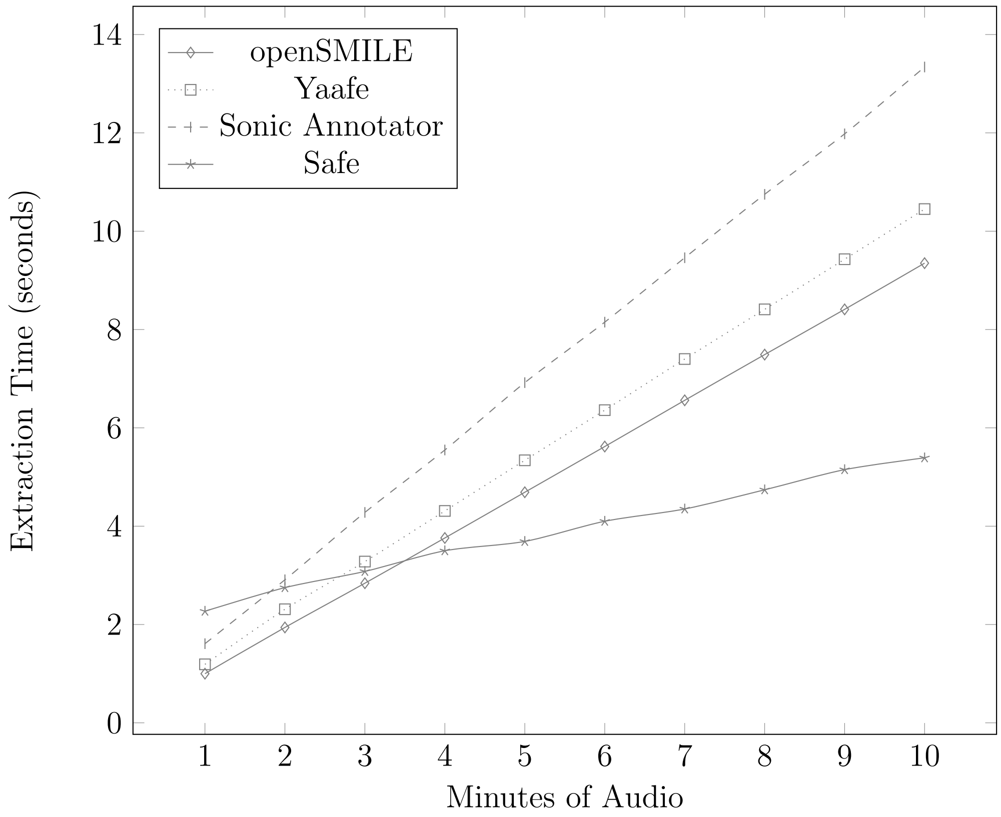


Figure 5.3: Feature extraction times for single audio files of increasing length

5.2 Dataflow Analysis

In order to test the benefits of dataflow analysis, Safe's performance was measured both with and without dataflow optimizations. Testing was performed using the same dataset and feature set described previously in section 5.1. For the sequential case, each audio file was loaded only once but the individual extraction algorithms were run independently with no shared calculations. This setup is very similar to the way many feature extraction libraries (including Sonic Annotator) work. Figure 5.4 shows the extraction times both with dataflow optimizations and without (sequential). For this particular feature set, we see a 33% decrease in the overall extraction time when dataflow optimizations are applied. However, the benefits can vary significantly depending on the selected feature sets and input parameters.

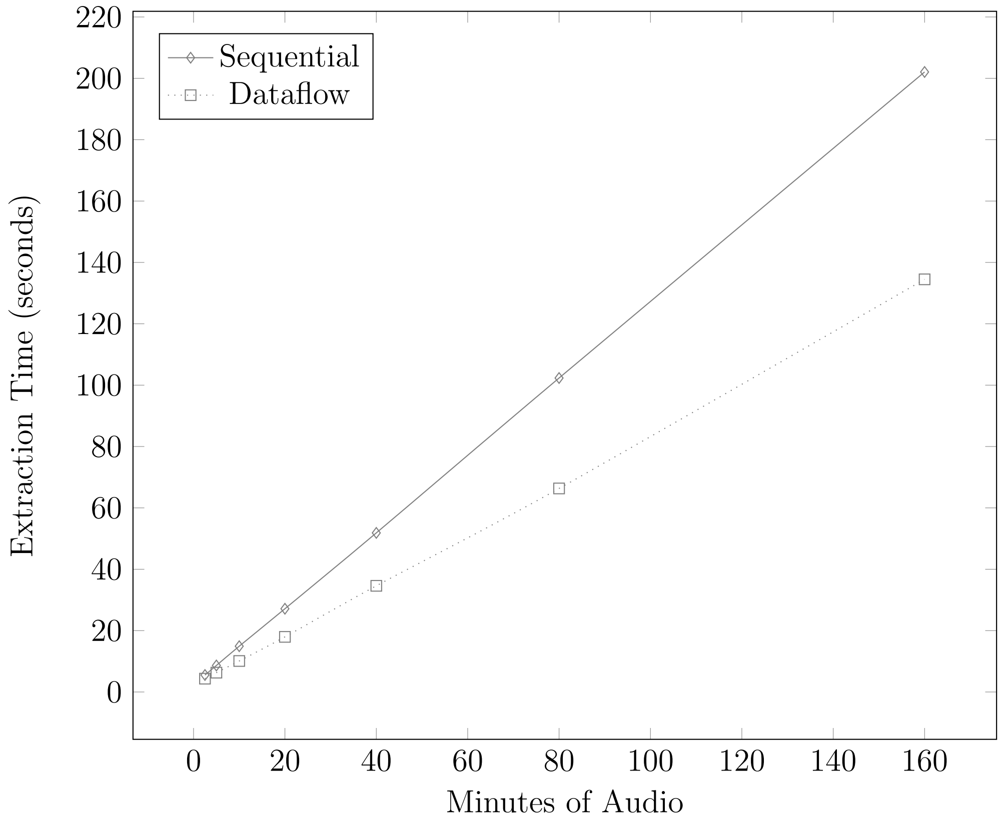


Figure 5.4: Performance increase from dataflow analysis (single-threaded)

5.3 Scalability Analysis

The techniques used to measure and describe scalability of the system are all based on Gunther's Universal Scalability Law (USL) [15]. Within this model, one of the primary measurements for parallelism in a system is the system's *speedup* factor. Speedup is defined as the ratio of elapsed time on a single processor T_1 to the elapsed time on p processors T_p.

$$S(p) = \frac{T_1}{T_p} \tag{5.2}$$

Speedup measurements are taken by executing a fixed workload on increasing numbers of processors (or machines). The result is a quantification of the reduced execution time for adding more processors to the system. Another important measurement of a system's parallelism, based on speedup, is *efficiency.* Efficiency measures the average speedup per processor, which is defined as:

$$E(p) = \frac{S(p)}{p} \tag{5.3}$$

In theory, a system that is perfectly parallel will have linear speedup T_1/p. In practice, however, there are other factors that make linear speedup in systems unrealistic. The Universal Scalability Model was developed to take these additional factors into account and give a realistic description of a systems true scalability. The two primary factors involved in this model are:

- *Contention* (σ) - The serial fraction of the total execution. This represents the portion of an algorithm that cannot be split up and needs to run serially.
- *Coherency* (κ) - The additional overhead of managing extra processors and interprocessor communication.

Taking these factors into account, the Universal Scalability equation for system capacity $C(p)$ is defined as:

$$C(p) = \frac{p}{1 + \sigma(p-1) + \kappa p(p-1)} \tag{5.4}$$

Equation 5.4 describes a *concave* function. Under this model, most systems will have a maximum capacity at p^*. Adding more processors beyond p^* will only degrade performance. The maximum capacity can be obtained through the following equation:

$$p^* = \left\lfloor \sqrt{\frac{1-\sigma}{\kappa}} \right\rfloor \tag{5.5}$$

The σ and κ parameters in the USL are calculated by performing regression analysis on the measured system throughput $X(p)$. The full procedure for calculating these parameters is outlined in [15].

All scalability testing of the system was performed on the University of Colorado at Colorado Springs virtual cloud infrastructure [10] running a VMWare vSphere 5 hypervisor. The particular cluster used consisted of 8 physical HP machines w/ 2.4GHz Intel Xeon E5530 CPUs. The total available resources in the cluster consisted of:

- 64 Processor Cores
- 153 GHz total CPU
- 512 GB Memory
- 55 TB Connected Storage

All virtual machines used in testing were running a 64-bit CentOS 6.4 operating system.

5.3.1 Multi-Processor Scalability

In addition to providing a simple model for developing concurrent applications, the Actor model provides a simple theoretical foundation for estimating scalability. Since each actor processes messages from its inbox serially, the contention factor σ in an actor system can be estimated by running the system with a single processor (or single-threaded) and measuring the percentage of time spent in each actor. The largest percentage of time spent in a single actor should be very close to the overall contention factor.

Using the same features and dataset described in section 5.1, figure 5.5 below shows the percentage of time spent computing each step (sub-feature) for different pool sizes. In the case where a single actor is dedicated to each sub-feature computation (pool size of 1), reading in the input audio and creating the frames takes approximately 21%

[10] http://www.uccs.edu/eas/eas_cloud.html

of the time on a single processor. Therefore, the theoretical serial contention factor σ_1 should be ≈ 0.21. In the current system, features that require aggregation (such as Spectral Flux) cannot be pooled and only a single actor is created regardless of pool size. Therefore, these features will often become the dominant factor in scalability when pooling is used. With this feature set, the Spectral Flux calculation takes approximately 13% of the time on a single processor. For pool sizes of two or more, the contention factor σ_n should be ≈ 0.13.

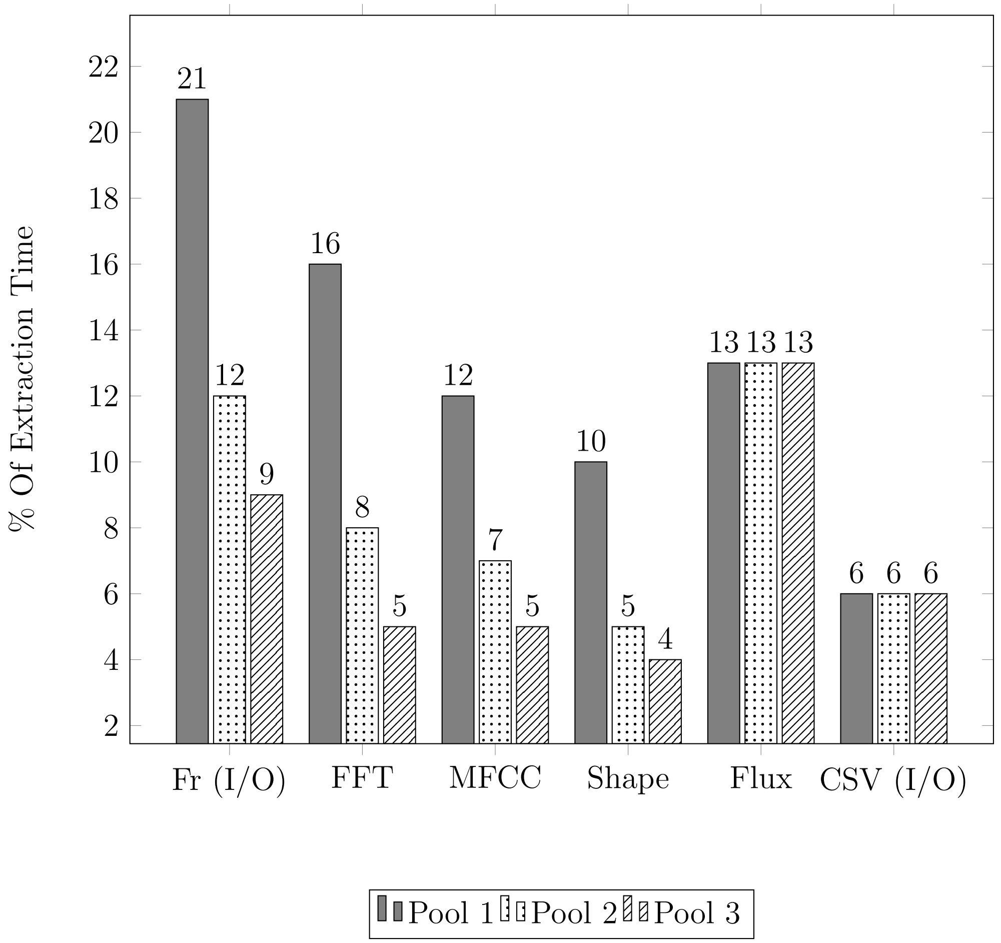


Figure 5.5: Percentage of total time taken by individual sub-feature actors

In order to test the system's actual multi-processor scalability, the same extraction test was run on virtual machines with an exponentially increasing number of single-core processors $p = 1, 2, 4, 8, \ldots, 64$. Multiple tests were run to capture the impact of increasing actor pool sizes. Figure 5.6 shows the measured system speedup. As expected from the theoretical model, no performance increases were seen for pool sizes greater than two.

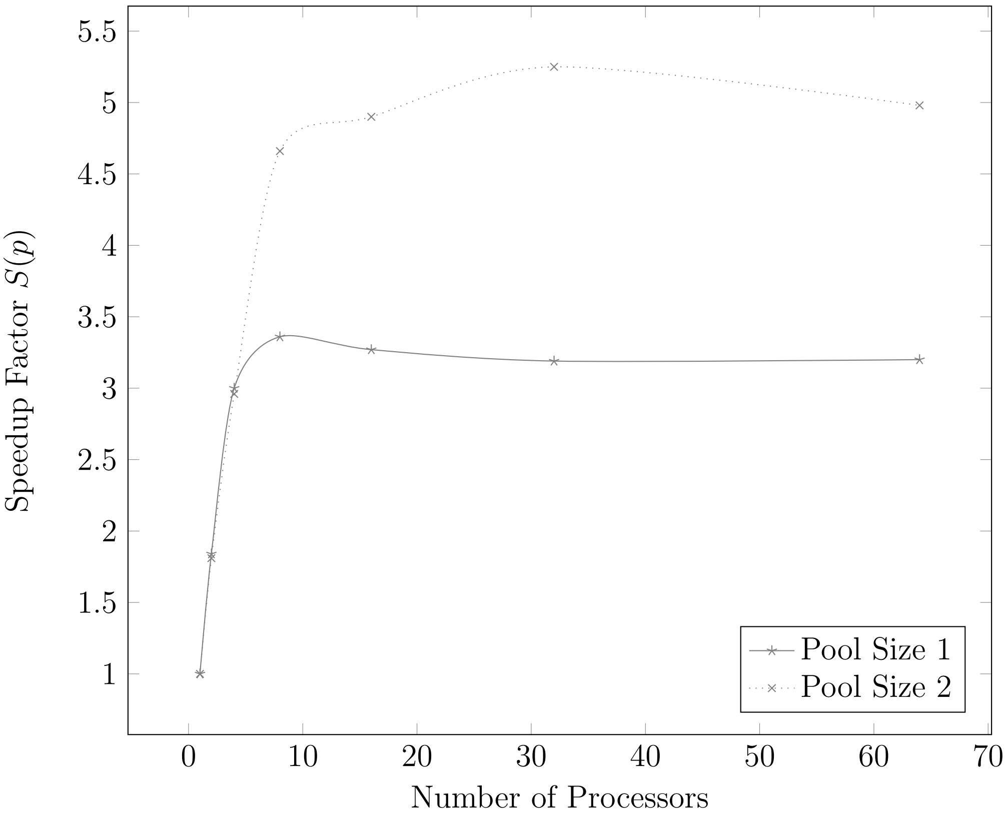


Figure 5.6: Feature extraction processor speedup

Figure 5.7 shows the measured throughput and theoretical curve fits based on the USL calculations. From these equation parameters, our maximum capacity p^* for this particular feature set is 20 processors for a pool size of one (single actor per sub-feature computation) and 28 processors for a pool size of two. The calculated contention factors $\sigma_1 = 0.203$ and $\sigma_2 = 0.123$ were very close to the theoretical expected values of 0.21 and 0.13.

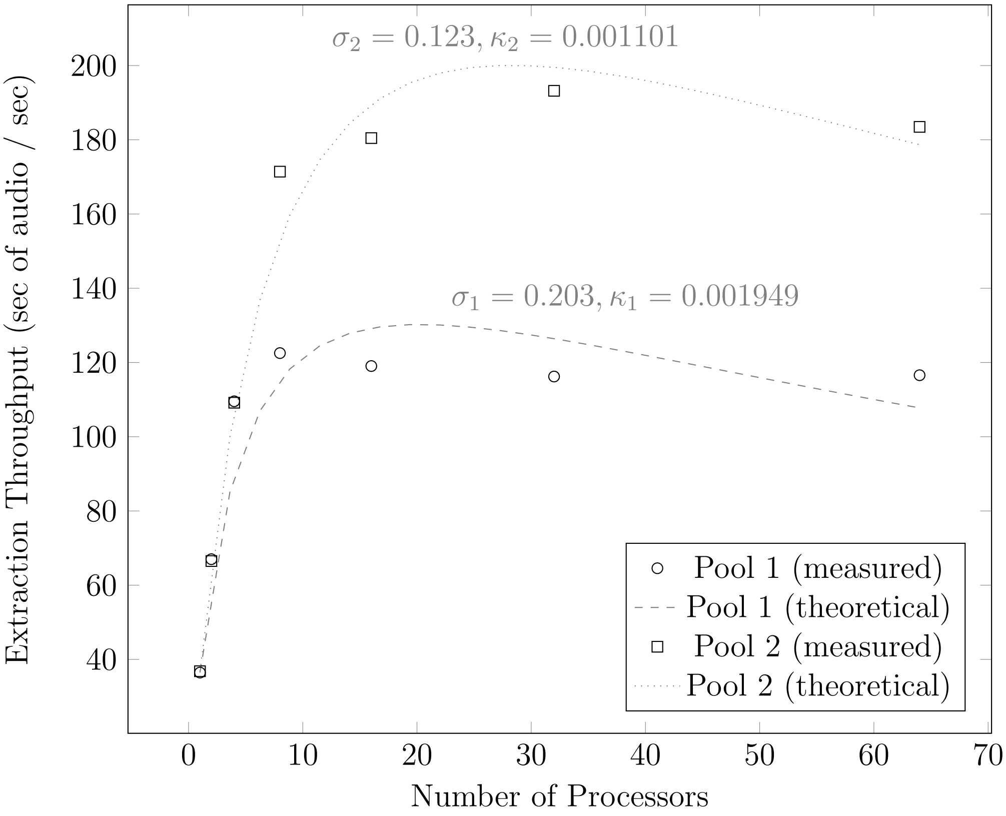


Figure 5.7: Feature extraction processor scalability

5.3.2 Multi-Node Cluster Scalability

The same USL model used to evaluate Safe's multi-processor scalability was used for measuring the distributed cluster scalability. Instead of increasing the number of processors on a single virtual machine, tests were run on increasing numbers of virtual machines. Each node (VM) in the cluster was given the same number of resources (processors, disk space, etc.). In most hosted VM environments, multiple VM's will often share and compete for physical resources such as disk space and processors. In an attempt to minimize this contention for these tests, each VM was mapped to a separate physical hard drive and processor. Since the physical cluster only provided six separate disk partitions and eight separate processors, the maximum number of nodes tested was six.

The first test, as outlined in section 4.3.1, relied on the dataset being distributed evenly across the cluster onto the local drives. For this test, the audio files were distributed across the cluster and placed in the same directory location on each node. Communication between nodes in this configuration is minimal and only relies on the initial broadcast message to perform extraction and the final aggregation step determining when all nodes have finished.

Figure 5.8 shows the measured and USL calculated theoretical scalability of this model. As expected, because of the minimal coordination between nodes, the scalability is close to linear. Based on the calculated contention and coherency, the theoretical maximum capacity p^* of this configuration is 82 nodes.

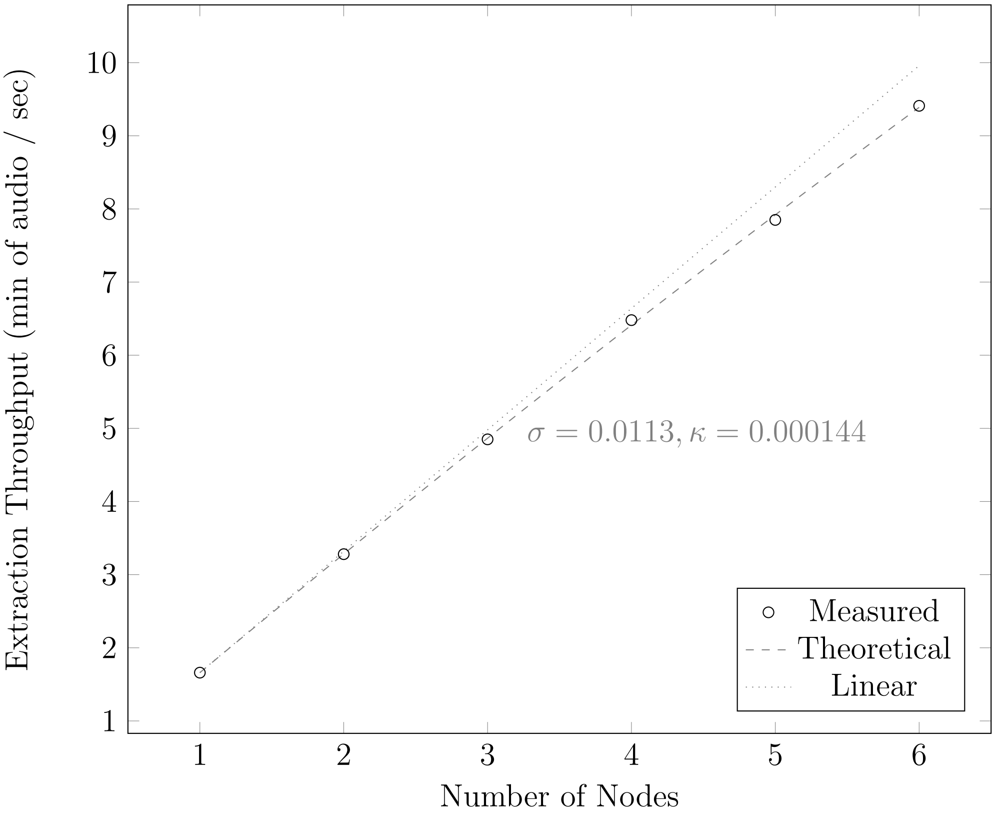


Figure 5.8: Feature extraction cluster/node scalability (local audio files)

While the previous localized data configuration has close to ideal scalability, it is a poor choice for fault-tolerance or flexibility in the distribution model. The second test, as described in 4.3.2, relied on the audio data being distributed through HDFS. The HDFS configuration for these tests specified a replication factor of 3 and a block size of 128 MB. For CD quality wav files (44.1 KHz, 16-bit stereo), 1 minute of audio is approximately equal to 10 MB. Therefore, most songs under 12 minutes in length were stored in a single block. Each node ran an instance of the Safe software and doubled as an HDFS data node. With each new node added, the cluster was re-balanced to more evenly distribute the audio files.

Initial testing leveraged a cluster-aware router that attempted to route extraction of individual files based on data locality. In this case, preference was given to nodes where the physical audio files and processing were co-located in an attempt to take advantage of HDFS short-circuit local reads. However, testing revealed that the HDFS re-balancing algorithm did not provide a truly even distribution of the data, causing the distributed workload to become unbalanced. Instead, a standard round-robin routing strategy was used which gave better overall performance.

Figure 5.9 below shows the measured and USL calculated scalability of the HDFS-based distribution. Not surprisingly, the contention and coherency factors are slightly higher in this configuration, but the fault-tolerance and flexibility make this a more appealing option for most systems. With this model, the maximum theoretical capacity p^* is 28 nodes.

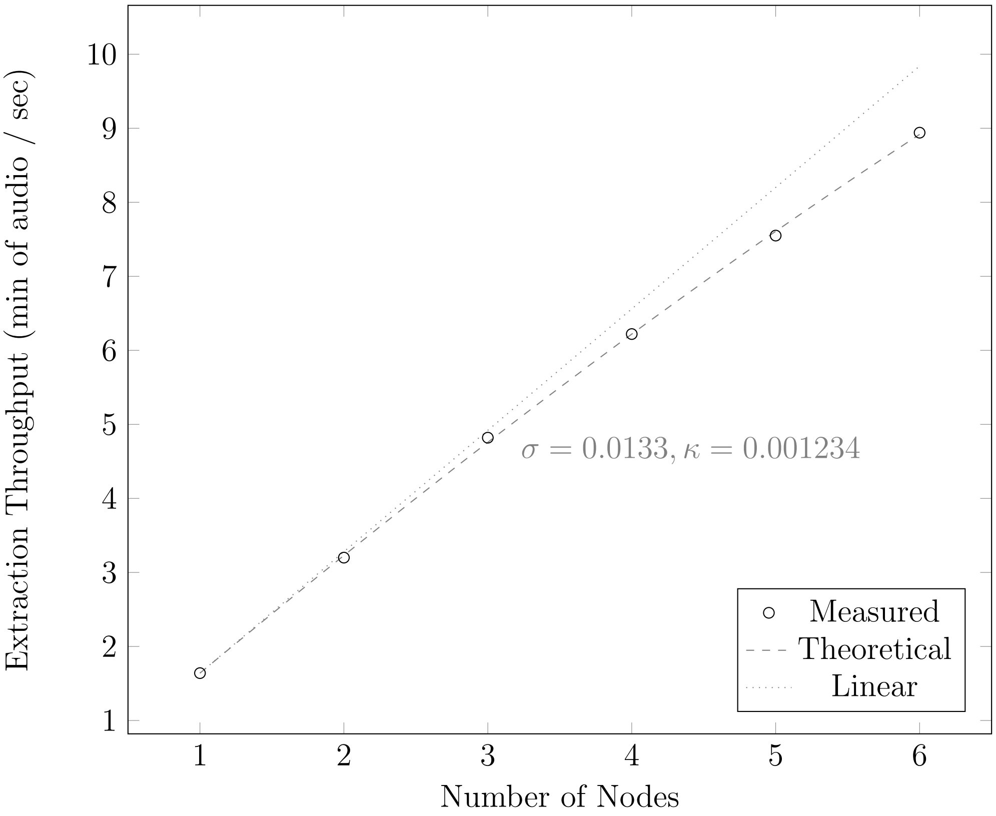


Figure 5.9: Feature extraction cluster/node scalability (HDFS distributed audio files)

Chapter 6

Conclusions

Many of the current frameworks for audio feature extraction have focused on algorithms and optimization techniques for serial execution. Dealing with larger datasets requires a more scalable model for extraction. In this paper, the Safe system for high-performance concurrent audio feature extraction was outlined. In comparison benchmarks with current leading MIR extraction libraries, Safe outperformed the existing solutions for all datasets with more than 3 minutes of audio.

Scalability analysis provided theoretical models for the system's multi-processor and multi-node cluster scalability. The actor model of concurrency used in Safe provides a simple theoretical model for predicting the vertical (processor) scalability with any given feature set. Further scalability testing demonstrated that the measured contention factors lined up with the expected theoretical values.

Two different models for multi-node cluster scalability were evaluated. The first model relied on the dataset being distributed across the cluster onto the individual node's local file systems. The second, more flexible and fault-tolerant model, leveraged a true distributed file system. Using the Universal Scalability Law to estimate the system's contention and coherency factors, the local file system based approach was shown to have close to ideal scalability. Although slightly less performant, the system's scalability using the distributed file system was still very strong.

All of the source code for Safe is available online [1] and released under the Eclipse Public License version 1.0 [2].

6.1 Future Work

In addition to adding more algorithms and features to Safe, there are certain areas where scalability of the system can be improved. Many of the current aggregation features in the system are un-necessarily limited to a single actor. In cases where aggregation is performed on a per-file basis, a smarter routing component can use the origin file information to appropriately route messages. More specifically, a consistent hashing router can be used that takes the origin file's path as the hash key.

All of the cluster scalability tests were performed on a homogenous cluster. Additional research into smarter distributed routing for unbalanced or inhomogeneous clusters would help in optimizing extraction times. Akka provides an adaptive probabilistic load balancing router that can be leveraged or extended for this task.

Lastly, future enhancements should address more aspects of fault-tolerance within the system. While minimal support for fault-tolerance is built into the system through the libraries being used, additional logic needs to be incorporated for handling failures during extraction. Ideally, the extraction system should be able to recover from node failures or network partitions.

[1] http://github.com/devonbryant/safe

[2] http://www.eclipse.org/org/documents/epl-v10.html

Appendix A

Implemented Features

A large portion of this research consists of analyzing MIR feature extraction algorithms and breaking them down into smaller sequential components for dataflow analysis. The following sub-sections describe the various feature extraction algorithms implemented in the system, along with any common or shared sub-feature computations.

A.0.1 Frame

The framing function produces an iterator of fixed-size, potentially overlapping, frames from an audio input stream. Each frame is normalized based on the encoding bit-depth and multiple channels are combined to produce a single frame of normalized audio. The following algorithm shows the combination for a single sample S based on the number of channels n and bit-depth d:

$$S = \frac{\sum_{i=1}^{n} s_i}{(2^d - 1)n} \tag{A.1}$$

A.0.2 Window

Safe provides a common set of windowing functions, including *Hann* (A.2), *Hamming* (A.3), *Blackman* (A.4), *Blackman-Harris* (A.5), and *Bartlett* (A.6) windows.

$$w(n) = 0.5\Big(1 - \cos\Big(2\pi\frac{n}{L-1}\Big)\Big), \quad 0 \le n < L \tag{A.2}$$

$$w(n) = 0.54 - 0.46\cos\Big(2\pi\frac{n}{L-1}\Big), \quad 0 \le n < L \tag{A.3}$$

$$w(n) = 0.42 - 0.5\cos\Big(2\pi\frac{n}{L-1}\Big) + 0.08\cos\Big(4\pi\frac{n}{L-1}\Big), \quad 0 \le n < L \tag{A.4}$$

$$\begin{gathered} w(n) = a_0 - a_1\cos\Big(\frac{2\pi n}{L-1}\Big) + a_2\cos\Big(\frac{4\pi n}{L-1}\Big) - a_3\cos\Big(\frac{6\pi n}{L-1}\Big), \quad 0 \le n < L \\ a_0 = 0.35875, a_1 = 0.48829, a_2 = 0.14128, a_3 = 0.01168 \end{gathered} \tag{A.5}$$

$$w(n) = \begin{cases} \frac{2n}{L-1} & 0 \le n \le \frac{L-1}{2} \\ 2 - \frac{2n}{L-1} & \frac{L-1}{2} \le n < L \end{cases} \tag{A.6}$$

A.0.3 Fast Fourier Transform

The FFT algorithm used in Safe is a JVM-optimized version of the standard *Cooley-Turkey* algorithm. The standard *Cooley-Turkey* algorithm is described as:

$$X_k = \sum_{m=0}^{N/2-1} x_{2_m} \mathrm{e}^{-\frac{2\pi i}{N/2}mk} + \mathrm{e}^{-\frac{2\pi i}{N}k} \sum_{m=0}^{N/2-1} x_{2_{m+1}} \mathrm{e}^{-\frac{2\pi i}{N/2}mk} \tag{A.7}$$

Further information on the implementation and optimization techniques used can be found online [1].

A.0.4 Power Spectrum

Safe provides various spectrum functions, including *Power* $P(k)$, *Magnitude* $A(k)$, and *Phase* $\phi(k)$. Each function is based on the *spectral coefficients* a_k and b_k computed by the FFT.

$$P(k) = a_k^2 + b_k^2 \tag{A.8}$$

$$A(k) = \sqrt{a_k^2 + b_k^2} \tag{A.9}$$

$$\phi(k) = \tan^{-1} \frac{b_k}{a_k} \tag{A.10}$$

A.0.5 Constant-Q Transform

The Constant-Q transform (CQT) is similar to the Fourier transform, but uses logarithmic spacing instead of linear spacing. This is beneficial in audio analysis since notes in western music are spaced logarithmically. The basic CQT algorithm is given by:

$$X_k = \frac{1}{N_k} \sum_{n=0}^{N_k-1} w_{k,n} x_n \mathrm{e}^{\frac{-j2\pi Qn}{N_k}} \tag{A.11}$$

The CQT algorithm used in Safe is based on the Brown and Puckette [5] algorithm, which leverages the standard FFT algorithm for faster computations. Figure A.1 shows the dataflow for the CQT algorithm.

[1] http://devonbryant.github.io/blog/2013/03/03/numerical-computing-with-scala/

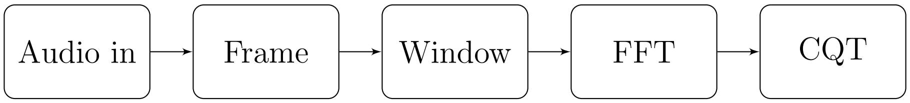


Figure A.1: Sequential dataflow of the Constant-Q transform

Name	**Description**	**Default Value**
sampleRate	Target (expected) sample rate of audio inputs	44100
stepSize	The step size (number of samples) for the framing function	512
windowType	Windowing function (bartlett, blackman, blackmanHarris, hamming, hann)	hann
binsPerOctave	The number of CQT bins/octave (default 24 = quarter-tone spacing)	24
maxFreq	Maximum frequency (hz) to look for (default is G10, MIDI note 127)	12543.854
minFreq	Minimum frequency (hz) to look for (default is C1, MIDI note 12)	16.351599
threshold	Minimum threshold	0.0054

Table A.1: CQT Parameters

A.0.6 Mel-Frequency Cepstral Coefficients

The *mel-frequency cepstrum* provides a description of an audio signals power spectrum based on the *mel scale*, a perceptually driven frequency scale. In this scale, the mel representation m of a frequency f (hz) is given by:

$$m = 1127 \log_e \left(1 + \frac{f}{700}\right) \tag{A.12}$$

The MFCC computation is based on the Davis and Mermelstein [7] algorithm, which is described as:

$$Y_k = \sum_{f=0}^{N/2-1} A(k) w_k(f) \tag{A.13}$$

$$MFCC_n = \sum_{k=0}^{K-1} \log Y_k \cos\left(\frac{(2k+1)n\pi}{2K}\right)$$

Where Y_k represents the application of K overlapping triangular (mel-scale spaced) filters on the magnitude spectrum of the signal $A(k)$ and n number of MFCC coefficients are calculated through a discrete-cosine transform on the log-energy outputs of Y_k. Figure A.2 shows the dataflow for the MFCC algorithm.

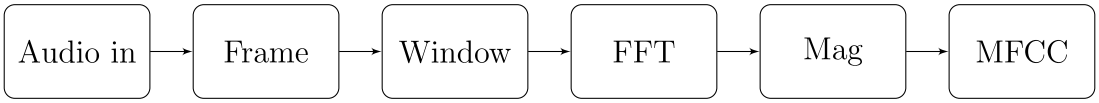


Figure A.2: Sequential dataflow for Mel-Frequency Cepstral Coefficients

Name	**Description**	**Default Value**
sampleRate	Target (expected) sample rate of audio inputs	44100
frameSize	Frame size (number of samples) for the framing function	1024
stepSize	Step size (number of samples) for the framing function	512
windowType	Windowing function (bartlett, blackman, blackmanHarris, hamming, hann)	hann
numCoeffs	Number of cepstral coefficients to extract	13
melFilters	Number of mel filter banks to use	40
minFreq	Minimum frequency (hz) for the filter bank	130.0
maxFreq	Maximum frequency (hz) for the filter bank	6854.0

Table A.2: MFCC Parameters

A.0.7 Spectral Shape

Spectral shape is an aggregate of four separate spectral features: *Centroid*, *Spread*, *Skewness*, and *Kurtosis*. The calculations for these features are described in [14], and are defined by the following equations:

$$S_{centroid} = \mu_1 \tag{A.14}$$

$$S_{spread} = \sqrt{\mu_2 - \mu_1^2} \tag{A.15}$$

$$S_{skewness} = \frac{2\mu_1^3 - 3\mu_1\mu_2 + \mu_3}{S_{spread}^3} \tag{A.16}$$

$$S_{kurtosis} = \frac{-3\mu_1^4 + 6\mu_1\mu_2 - 4\mu_1\mu_3 + \mu_4}{S_{spread}^4} - 3 \tag{A.17}$$

Where $\mu_i = \frac{\sum_{k=0}^{N-1} k^i A(k)}{\sum_{k=0}^{N-1} A(k)}$ and $A(k)$ represents the magnitude spectrum of the signal. Figure A.3 shows the dataflow sequence for the spectral shape algorithm.

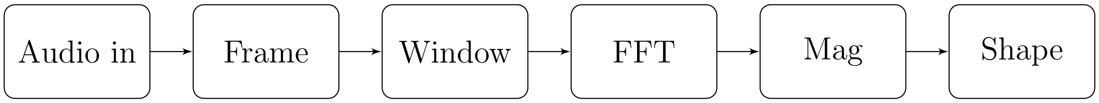


Figure A.3: Sequential dataflow for Spectral Shape

Name	Description	Default Value
sampleRate	Target (expected) sample rate of audio inputs	44100
frameSize	Frame size (number of samples) for the framing function	1024
stepSize	Step size (number of samples) for the framing function	512
windowType	Windowing function (bartlett, blackman, blackmanHarris, hamming, hann)	hann

Table A.3: Spectral Shape Parameters

A.0.8 Spectral Flux

The spectral flux algorithm is based on the algorithm for measuring change in the power spectrum described by Dixon in [11]. The algorithm computes the flux between consecutive frames (n, $n-1$) as the summation of the positive differences between the magnitude at each frequency bin, as shown in the following equation.

$$SF(n) = \sum_{k=0}^{N-1} H\Big(|X(n,k)| - |X(n-1,k)|\Big) \tag{A.18}$$

In this equation, H represents a half-wave rectifier function where $H(x) = \frac{x+|x|}{2}$. Figure A.4 below shows the dataflow for the spectral flux extraction algorithm.

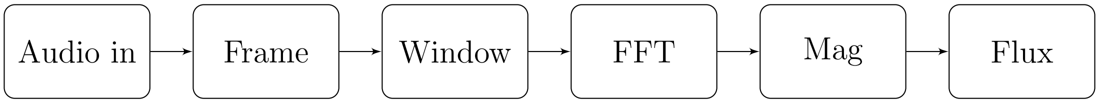


Figure A.4: Sequential dataflow for Spectral Flux

Name	Description	Default Value
sampleRate	Target (expected) sample rate of audio inputs	44100
frameSize	Frame size (number of samples) for the framing function	1024
stepSize	Step size (number of samples) for the framing function	512
windowType	Windowing function (bartlett, blackman, blackmanHarris, hamming, hann)	hann
diffLength	Compare frames spaced n length apart, 1 = consecutive frames	1

Table A.4: Spectral Flux Parameters

A.0.9 Onset Detection

Safe implements the spectral onset detection described in [3]. The algorithm extends the standard spectral flux calculation by applying a logarithmically spaced filter-bank (similar to the Constant-Q transform spacing). The onset detection function is given by:

$$O(n) = \sum_{k=1}^{k=N/2} H\Big(|X_{filt}^{log}(n,b)| - |X_{filt}^{log}(n-1,b)|\Big) \tag{A.19}$$

Peaks in the onset detection function are selected if all of the following conditions are satisfied:

1. $ODF(n) = max(ODF(n - w_1 : n + w_2)$
2. $ODF(n) \geq mean(ODF(n - w_3 : n + w_4)) + \delta$
3. $n - n_{last\_onset} > w_5$

Where δ is a fixed threshold value and $w_1, w_2, \ldots, w_5$ are various tunable parameters. Figure A.5 shows the dataflow for the onset detection algorithm.

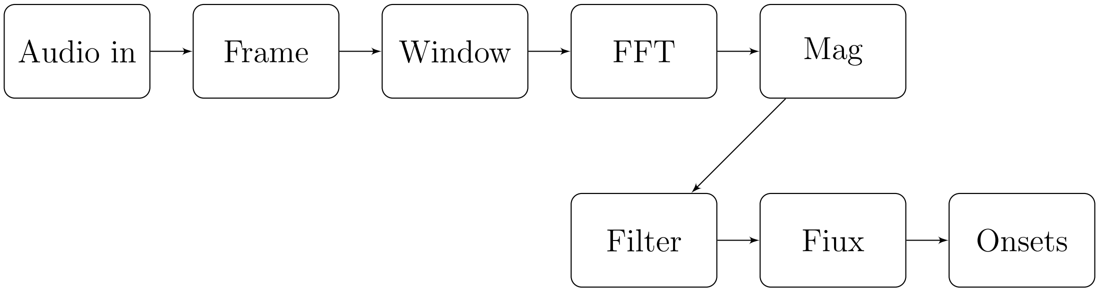


Figure A.5: Sequential dataflow for Spectral Onsets

Name	**Description**	**Default Value**
sampleRate	Target (expected) sample rate of audio inputs	44100
frameSize	Frame size (number of samples) for the framing function	1024
stepSize	Step size (number of samples) for the framing function	512
windowType	Windowing function (bartlett, blackman, blackmanHarris, hamming, hann)	hann
ratio	Minimum activation ratio for windowing function	0.22
threshold	Minimum threshold (δ) for peak-picking	2.5

Table A.5: Spectral Onset Parameters

References

[1] Gul Abdulnabi Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, 1985.

[2] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[3] Sebastian Böck, Florian Krebs, and Markus Schedl. Evaluating the online capabilities of onset detection methods. In *ISMIR*, pages 49–54, 2012.

[4] Stuart Bray and George Tzanetakis. Distributed audio feature extraction for music. In *ISMIR*, pages 434–437, 2005.

[5] Judith C Brown and Miller S Puckette. An efficient algorithm for the calculation of a constant q transform. *The Journal of the Acoustical Society of America*, 92:2698, 1992.

[6] Chris Cannam, Michael O. Jewell, Christophe Rhodes, Mark Sandler, and Mark d'Inverno. Linked data and you: Bringing music research software into the semantic web. *Journal of New Music Research*, 39(4):313–325, 2010.

[7] Steven Davis and Paul Mermelstein. Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 28(4):357–366, 1980.

[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[9] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, volume 7, pages 205–220, 2007.

[10] François Deliège, Bee Yong Chua, and Torben Bach Pedersen. High-level audio features: Distributed extraction and similarity search. In *ISMIR*, pages 565–570, 2008.

[11] Simon Dixon. Onset detection revisited. In *Proceedings of the 9th International Conference on Digital Audio Effects*, volume 120, pages 133–137, 2006.

[12] Florian Eyben, Martin Wöllmer, and Björn Schuller. Opensmile: the munich versatile and fast open-source audio feature extractor. In *Proceedings of the international conference on Multimedia*, pages 1459–1462. ACM, 2010.

[13] Roy T Fielding and Richard N Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, 2002.

[14] Olivier Gillet and Gaël Richard. Automatic transcription of drum loops. In *Acoustics, Speech, and Signal Processing, 2004. Proceedings.(ICASSP'04). IEEE International Conference on*, volume 4, pages iv–269. IEEE, 2004.

[15] Neil J. Gunther. *Guerrilla Capacity Planning*. Springer Berlin Heidelberg, 2007.

[16] Naohiro Hayashibara, Xavier Defago, Rami Yared, and Takuya Katayama. The φ accrual failure detector. In *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*, pages 66–78. IEEE, 2004.

[17] Gregor Hohpe and Bobby Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004.

[18] Olivier Lartillot, Petri Toiviainen, and Tuomas Eerola. A matlab toolbox for music information retrieval. In *Data analysis, machine learning and applications*, pages 261–268. Springer, 2008.

[19] Benoit Mathieu, Slim Essid, Thomas Fillon, Jacques Prado, and Gaël Richard. Yaafe, an easy to use and efficient audio feature extraction software. In *ISMIR*, pages 441–446, 2010.

[20] Daniel McEnnis, Cory McKay, and Ichiro Fujinaga. Overview of omen. In *ISMIR*, pages 7–12, 2006.

[21] Daniel McEnnis, Cory McKay, Ichiro Fujinaga, and P Depalle. jaudio: Additions and improvements. In *ISMIR*, pages 385–386, 2006.

[22] Cory McKay, Ichiro Fujinaga, and Philippe Depalle. jaudio: A feature extraction library. In *Proceedings of the International Conference on Music Information Retrieval*, pages 600–3, 2005.

[23] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[24] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, September 2005.

[25] George Tzanetakis and Perry Cook. Marsyas: A framework for audio analysis. *Organised sound*, 4(3):169–175, 2000.

[26] Robbert Van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. In *Middleware'98*, pages 55–70. Springer, 1998.