IMPLICIT PATCHING FOR DATAFLOW-BASED AUDIO ANALYSIS AND SYNTHESIS

Stuart Bray and *George Tzanetakis
University of Victoria
Computer Science Department *(Also in Music)

ABSTRACT

Programming software for audio analysis and synthesis is challenging. Dataflow-based approaches provide a declarative specification of computation and result in efficient code. Most practitioners of computer music are familiar with some form of dataflow programming where audio applications are constructed by connecting components with "wires" that carry data. Examples include networks of unit generators in Music-V style languages and visual patches in Max/Msp or PD. Even though existing dataflow-based audio systems offer a concise conceptual model of signal computation, this model does have limitations. In many cases, these limitations are a consequence of the programmer having to explicitly specify connections between components. Two such limitations are the difficulty of handling spectral data and the need for fixed-size buffers between components. In this paper we introduce *Implicit Patching* (IP), a dataflow-based approach to audio analysis and synthesis that attempts to address these limitations. By extending dataflow semantics a large number of connections are automatically created and buffer sizes can be changed dynamically. The resulting model is also particularly suited for distributed systems. We describe *Marsyas-0.2*, a software framework based on IP, and provide examples that illustrate the strengths and limitations of the proposed approach.

1. INTRODUCTION

There is a plethora of programming languages, frameworks and environments for the analysis and synthesis of audio signals. The processing of audio signals requires extensive numerical calculations over large amounts of data especially when real-time performance is desired. Therefore efficiency has always been a major concern in the design of audio analysis and synthesis systems. Dataflow programming is based on the idea of expressing computation as a network of processing nodes/components connected by a number of communication channels/arcs. Computer Music is possibly one of the most successful application areas for the dataflow programming paradigm. The origins of this idea can possibly be traced to the physical re-wiring (patching) employed for changing sound characteristics in early modular analog synthesizers. From the pioneering work on unit generators in the Music *N* family of language to currently popular visual programming environments such as Max/Msp and Pure Data (PD), the idea of patching components to build systems is familiar to most computer music practitioners.

Expressing audio processing systems as dataflow networks has several advantages. The programmer can provide a declarative specification of what needs to be computed without having to worry about the low level implementation details. The resulting code can be very efficient and have low memory requirements as data just "flows" through the network without having complicated dependencies. In addition, dataflow approaches are particularly suited for visual programming. One of the initial motivation for dataflow ideas was the exploitation of parallel hardware and therefore dataflow systems are particularly suited for parallel and distributed computation.

Despite these advantages, dataflow programming has not managed to become part of mainstream programming and replace existing imperative, object-oriented and functional languages. Some of the traditional criticisms aimed at dataflow programming include: the difficulty of expressing complicated control information, the restrictions on using assignment and global state information, the difficulty of expressing iteration and complicated data structures, and the challenge of synchronization.

There are two main ways that existing successful dataflow systems overcome these limitations. The first is to embed dataflow ideas into an existing programming language. This is called coarse-grained dataflow in contrast to fine-grained dataflow where the entire computation is expressed as a flow graph. With coarse-grained dataflow, complicated data structures, iteration, and state information are handled in the host language while using dataflow for structured modularity. The second way is to work on a domain whose nature and specific constraints are a good fit to a dataflow approach. For example, audio and multimedia processing typically deals with fixed-rate calculation of large buffers of numerical data.

Computer music has been one of the most successful cases of dataflow applications even though the academic dataflow community doesn't seem to be particularly aware of this fact. Existing audio processing dataflow frameworks have difficulty handling spectral and filterbank data in an conceptually clear manner. Another problem is the restriction of using fixed buffer sizes and therefore fixed audio and control rates. Both of these limitations can be traced to the restricted semantics of patching as well as the

need to explicitly specify connections. *Implicit Patching* the technique described in this paper is an attempt to overcome these problems while maintaining the advantages of dataflow computation. *Marsyas-0.2* is a software framework for audio analysis and synthesis described in this paper that is structured around the idea of IP. In order to illustrate the concept of IP we provide specific examples from audio analysis, synthesis and distributed computation.

2. RELATED WORK

Dataflow programming has a long history. The original (and still valid) motivation for reasearch into dataflow was to take advantage of parallelism. Motivated by criticisms of the classical von Neumann hardware architecture such as [1, 2] dataflow architectures for hardware were proposed as an alternative in the 1970s and 1980s. During the same period a number of textual dataflow languages such as Lucid [3] were proposed. Despite expectations that dataflow architectures and languages would take over from von Neumann concepts this didn't happen. However during 1990s there was a new direction of growth in the field of dataflow visual programming languages especially in specific application domains. Succesful commercial examples include Labview [1] and SimuLink [2]. A recent comprehensive review of the history of dataflow programming languages can be found in [4]. Another recent trend has been to view dataflow computation as a software engineering methodology for building systems using existing programming languages [5, 6].

It is interesting to note that the use of dataflow ideas in Computer Music follows a similar trajectory. Initial experimentation started in the 1960-1970s with modular analog synthesizers that could be programmed by physically "patching" wires [7]. Textual dataflow programming languages in Computer Music are exemplified by the long legacy of the Music *N* family whose most well known and popular member today is Csound [8]. Today the use of visual dataflow programming environments such as Max/MSP and Pure Data (PD) is pervasive in the computer music community [9, 10]. An object-oriented metamodel for digital signal processing that abstracts many of the dataflow ideas used in audio and music processing is presented in [11]. The use of object composition advocated in this paper has similar advantages to the use of expressions in [12].

Marsyas 0.2, the software framework for audio analysis and synthesis described in this paper, evolved from *Marsyas 0.1* [13] a framework that focused mostly on audio analysis and Music Information Retrieval. One of motivating factors for the rewrite of the code and architecture was the desire to add audio synthesis capabilities influenced by the design of the Synthesis Toolkit [14]. Other influences include the powerful but complicated flow architecture of CLAM [11], the interesting patching model of Chuck [15] and ideas from Aura [16]. The matrix model used in *Implicit Patching* was influenced by the design of SDIFF [17] and the naming scheme for controls is inspired by the Open Sound Control (OSC) format [18]. The code structure was motivated by design patterns [19].

Implicit patching, the technique described in this paper is illustrated with examples from audio analysis and synthesis. The phasevocoder [20] is a well known computationally intensive audio synthesis algorithm. From audio analysis we show how a standard audio feature extraction front-end such as the ones described in [21] can be expressed in *Marsyas-0.2*.

3. MARSYAS-0.2 ARCHITECTURE

Marsyas-0.2 [3] is a software framework, written in C++, for rapid prototyping and experimentation with audio analysis and synthesis with specific emphasis on processing music signals. The main goal is to provide a general, extensible and flexible framework that allows easy experimentation with algorithms and provides the fast performance necessary for developing real time audio analysis and synthesis tools. A variety of existing building blocks that form the basis of many published algorithms are provided as dataflow components that can be composed to form more complicated algorithms (black-box functionality). In addition, it is straightforward to extend the framework with new building blocks (white-box functionality). The goal of this section is not to provide an extensive overview of the system architecture but provide the necessary context to understand the ideas described in the paper.

In *Marsyas* terminology the processing nodes of the dataflow network are called *MarSystems* and provide the basic building blocks out of which more complicated systems are built. As will be shown in the next section essentially any audio processing can be expressed as a large composite *MarSystem* which is assembled by appropriately connected basic *MarSystems*. Some representative *MarSystems* provide in *Marsyas-0.2* are the following:

- Input/Output (Sources and Sinks)
  - Soundfile I/O for .wav,.au and .mp3 files
  - Real-time audio I/O using RtAudio
  - Matlab, Weka, Octave I/O
- Feature Extraction
  - Short-Time Fourier Transform
  - Discrete Wavelet Transform
  - Centroid, Rollloff, Flux, Contrast
  - Mel-Frequency Cepstral Coefficients
  - Auditory filterbanks
- Synthesis
  - Wavetable synthesis
  - FM synthesis

[1] `http://www.ni.com/labview/`

[2] `http://www.mathworks.com/products/simulink/`

[3] `http://marsyas.sourceforge.net`

  - Phasevocoder
- Machine Learning
  - Gaussian Mixture Model classifier
  - K-nearest neighbor classifier
  - Principal Component Analysis
  - K-Means clustering

In addition to being able to process data, in order for *MarSystems* to be useful, they need additional information. For example a *SoundFileSource* needs to the name of the soundfile to be opened and a *Gain* could be adjusted while data is flowing through. This is accomplished by a separate message passing mechanism. Therefore, similarly to CLAM [11], *Marsyas-0.2* makes a clear distinction between data-flow which is synchronous and control-flow which is asynchronous. Because *MarSystems* can be assembled hierarchically the control mechanism utilizes a path notation similar to OSC [18]. For example *Series/playbacknet/Gain/g1/real/gain* is the control name for accessing the gain control of a *Gain MarSystem* named *g1* in a *Series* composite named *playbacknet*. A mechanism for linking top-level controls to the longer full path control names is provided. For example a single gain control at the top-level can be linked to the gain controls of 20 oscillators in a synthesis instrument. That way one-to-many mappings can be achieved in a similar way to the use of regular expressions in OSC [18].

Dataflow in *Marsyas-0.2* is synchronous which means that at every "tick" a specific slice of data is propagated across the entire dataflow network. This eliminates the need for queues between processing nodes and enables the use of shared buffers which improves performance. This is similar to the way Unix pipes are implemented.

One of the most useful characteristics of *MarSystems* is that they can be instantiated at run-time. Because they are hierarchically composable that means that any complicated audio computation expressed as a dataflow network can be instantiated at run-time. For example multiple instances of any complicated network can be created as easily as the basic primitive *MarSystems*. This is accomplished by using the *Prototype* and *Composite* design patterns [19].

The combination of runtime instantiation and composability enables declarative specification of any dataflow network without any code compilation required. This provides a lot of the flexibility of interpreted languages while still retaining the fast performance of compiled code running inside each *MarSystem*. The user only needs to actually compile source code when adding a new *MarSystem* (creating a new type of processing object) or when trying to perform some computation that can not be expressed easily as a dataflow network. An alternative view of runtime instantiation is that it is similar to audio plugins such as VST that are hierarchically composable into networks.

Currently there are three ways to build audio analysis and applications in *Marsyas-0.2*. The first is the traditional method of writing directly C++ code and compiling

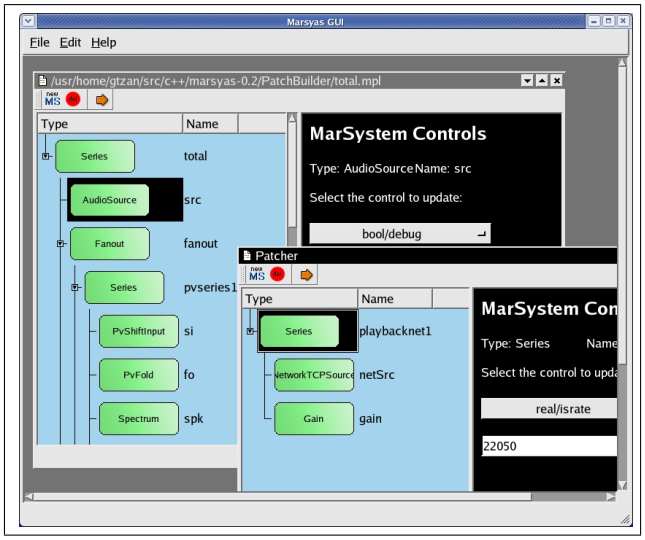

Figure 1. Marsyas-0.2 Visual Patch Builder

an executable. The second is based on a simple scripting language that provides syntactic constructs for building the dataflow network, setting appropriately the controls and moving sound through the network. The third way is to use a visual patch builder which uses the scripting language "under the hood". The following code shows how a simple network for soundfile playback can be specified in the scripting language (with comments):

```
# create instances
SoundFileSource src, Gain g1
AudioSink dest, Series pnet

# add MarSystems to Series Composite
src, g1, dest > pnet

# update control to play at half volume
Series/pnet/Gain/g1/real/gain 0.5

# hear the sound
run pnet
```

EXAMPLE 1

Figure 1 shows the visual patchbuilder that can be used for specifying dataflow networks and controls. It is our belief that a powerful audio analysis and synthesis framework should support both visual and textual programming of the same underlying system. An example of such an approach is described in [22].

4. IMPLICIT PATCHING

The basic idea behind *Implicit Patching* is to use object composition rather than explicitely specifying connections between input and output ports in order to construct the dataflow network. For example the following pseudo-code examples illustrates the difference of *Explicit* and *Implicit Patching* in a simple playback network.

```
# EXPLICIT PATCHING
create source, gain, dest
# connect the appropriate in/out ports
connect(source.out1, gain.in1);
connect(gain.out1, dest.in1);

# IMPLICIT PATCHING
create source, gain, dest
# create a composite that
# is essentially the network
create series(source, gain, dest);
```

EXAMPLE 2

The idea of *Implicit Patching* evolved from the integration of three different ideas that were developed independently in previous versions of Marsyas. These three ideas and how they are integrated are described bellow and in the following section examples illustrating the expressive power of *Implict Patching* are presented.

The first idea originated from the desire not to be constrained to fixed buffer sizes and to have proper semantics for spectral data. The majority of existing audio processing environments require that all processing objects in a flow network/visual patch, process fixed size buffers of audio samples (typical numbers are 64 or 128 samples). Having fixed buffer sizes simplifies memory management and simplifies patching as all connections are treated the same way. However, some applications like audio feature extraction require a variety of different buffer sizes to flow through the network (for example feature vectors typically have much lower dimensionality than audio data). Even though it is possible to have dynamic buffer sizes in explicit patching systems it is complex to implement and frequently requires a lot of work from the programmer to appropriately set the connections. In addition, these fixed size buffers are reused for holding spectral data and it is up to the programmer to correctly connect the spectral data to objects that process such data. The result is that the exact details of the Short Time Fourier Transform are encapsulated as a black box and the programmer has little control over the process. Our proposed solution to these two problems is to extend the semantics of the data that is processed. In Marsyas-0.2, processing objects (*MarSystems*) process chunks of data called *Slices*. *Slices* are matrices of floating point numbers characterized by three parameters: number of samples (things that "happen" at different instances in time), number of observations (things that "happen" at the same time instance) and sampling rate. This approach is similar to the matrix approach used in the Sound Description Interchange Format (SDIF) [17].

Figure 2 shows a *MarSystem* for spectral processing that converts an incoming audio buffer of 512 samples of 1 observation at a sampling rate of 22050 to 1 samples of 512 observations (the FFT bins) at a lower sampling rate of 22050/512. By propagating information about the sampling rate and the number of observations through the dataflow network, the use of *Slices* provides more correct and flexible semantics for spectral processing and feature extraction. *MarSystems* are designed so that they can handle *Slices* with arbitrary dimensions with one important constraint: they need to be able to calculate their *Slice* output parameters *Slice* from their *Slice* input parameters. For example it is possible to change the input number of samples to the *MarSystem* shown in Figure 2 to 1024 and the *MarSystem* will automatically determine that the number of observations of the output *Slice* is also 1024.

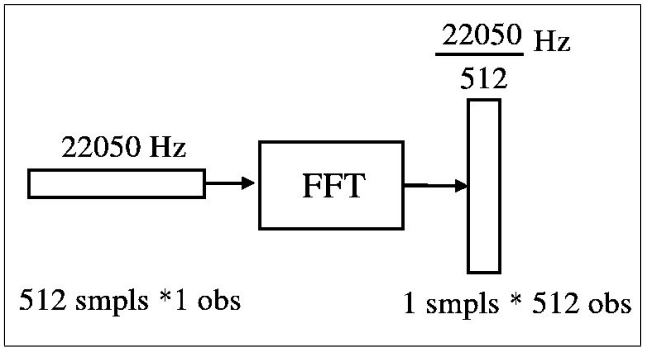

Figure 2. MarSystem and corresponding slices for spectral processing

The second major idea behind *Implict Patching* is the use of *Composite* design pattern [23] as a mechanism for constructing dataflow networks. The extended semantics of *Slices* require careful manipulation of buffer sizes especially if run-time changes are desired. The first composite used was *Series* which connects a list of *MarSystems* in series so that the output of the first one becomes the input to the second etc (similar to Unix pipe mechanism). The pseudo-code Example 2 above uses a *Series* composite. Initially composites were used a programming shortcuts. However, gradually we discovered that they offer many advantages and we decide to made them the main mechanism for constructing complicated *MarSystems* out of simpler ones. Their advantages include hierarchical encapsulation, automatic dynamic handling of all internal buffers, and run-time instantiation. More specifically, any dataflow network, no matter how complicated, is represented as a single *MarSystem* hierarchically composed of other simpler *MarSystems*, multiple instances of any *MarSystems* can be instantiated at run-time and all internal patching and memory handling is encapsulated.

The third idea was the unification of *Sources* and *Sinks* as regular *MarSystems* that have both input and output. *Sources* are processing objects that have only output and *Sinks* only have input. In order to be able to use them as any *MarSystem* we extend them in the following way: *Sources* mix their output with their input and *Sinks* propagate their input to their output while at the same time playing/writing their input. This way, for example, one can connect a *SoundFileSink* to a *AudioSink* and the data will be written both to a sound file and played using the audio device. Basically in both *Sources* and *Sinks* data gets injected into the network as a side effect but they can be used anywhere inside a network.

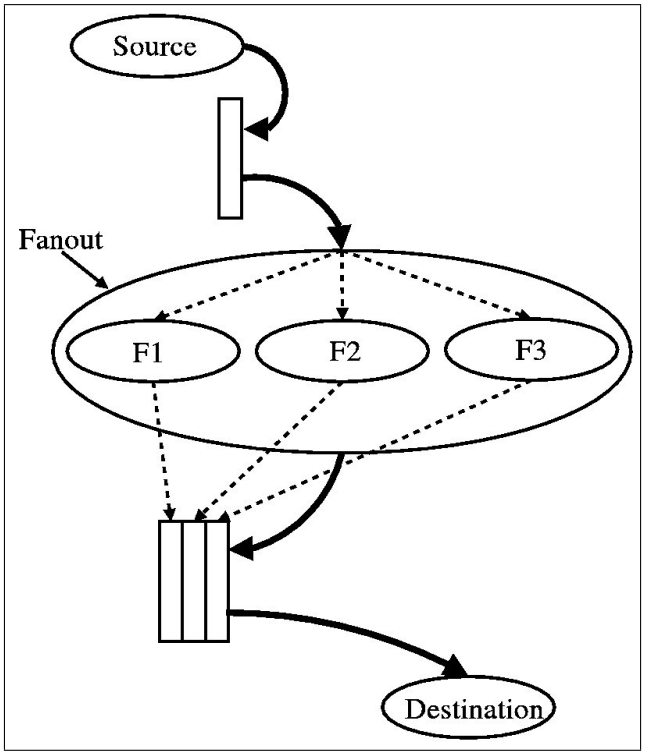

Figure 3. Fanout composite with buffers

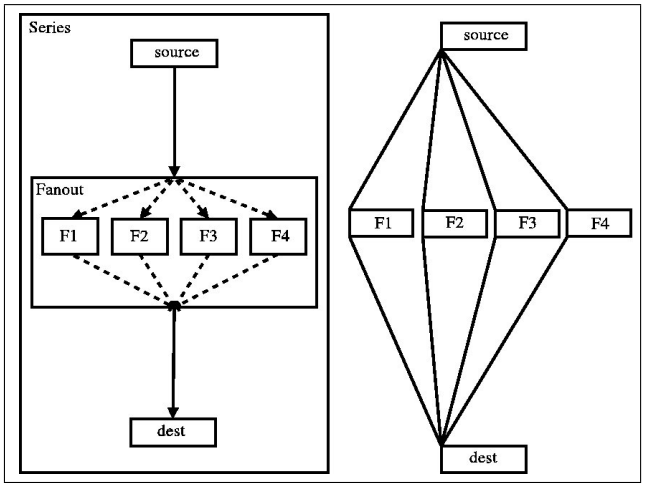

Figure 4. Comparison of Implicit Patching (left) and Explicit Patching (right)

Implicit Patching is made feasible by the integration of these three ideas. In this approach, each *MarSystem* has only one input port and one output port and consumes/produces only one token. However because of the extended semantics of *Slices* one can essentially have multiple input/output ports (as observations) and consume/produce multiple tokens (as samples). This enables non-trivial *Composites* such as *Fanout* to be created. That way, the expressive power of composition is increased and a large variety of complex dataflow networks can be expressed only using object composition and therefore no *Explicit Patching*.

To illustrate this approach, consider the *Fanout* composite which takes as input a slice and is built from a list of *MarSystems*. The input slice is then used as input to each internal *MarSystem* and their outputs are stacked as observations in the output *Slice* of the *Fanout*. For example a filterbank can be easily implemented as a *Fanout* where each filter is a internal component *MarSystem*. The filterbank *MarSystem* will take as input a slice of *N* samples by *1* observations and write to an output slice of *N* samples by *M* observations, where *M* is the number of filters. Because the innter loops of *MarSystems* iterate over both samples and observations if we connect the filterbank with, for example, a *Normalize MarSystem* each row of samples corresponding to a particular observation (each channel of the filterbank) will be normalized appropriately. This can be extremely handy in large filterbanks as the part of the network after the filterbank doesn't need to know how many filter outputs are produced. This information is taken implicitely from the number of observations. Figure 3 shows graphically how *Slices* are used in a *Fanout*. The dotted lines show the patching that is done implicitely by the *Fanout*. The black arrows show the main flow again created implicitely by a *Series* composite. In contrast, in environments with explicit patching such as Max/MSP each connection between the filters of the filterbank and the input would have to be created by the user. Figure 4 shows the difference between *Implicit Patching* (left) where the dotted lines are created automatically from the semantics of compositing and *Explicit Patching* (right) where each connection must be created separately. Even though environments such as Max/MSP or PD provide subpatching the burden of internal patching is still on the user.

We conclude this section with a non-trivial example illustrating the expressive power of *Implicit Patching*. Figure 5 shows how a layer of nodes in an Artificial Neural Network can be expressed using a *Fanout*. The input to the layer (the output of the previous layer) consists 4 numbers x_1, x_2, x_3, x_4. These 4 numbers (observations) based on the *Fanout* semantics become the input to each individual neuron (N_i) of the layer. Each neuron forms a weighted sum (with weights specific to each neuron) of the input, applies a sigmoid function to the sum and outputs a single output. The outputs using the *Fanout* semantics are stacked as observations y_1, y_2, y_3 (one for each neuron) ready for processing for the next layer. Figure 5 illustrates this process graphically (left side) and contrasts it with explicit patching (right side). In *Marsyas-0.2*, creating an artificial neural network using an annNode *MarSystem* is simply a series of fanouts of annNodes (more specifically seriesNet(fanoutLayer1, fanoutLayer2, ..., fanoutLayerM) where fanoutLayer1(annNode11, annNode12, ..., annNode1N). All the connections are created implicitely.

5. EXAMPLES

In this section, examples from audio analysis, synthesis and distributed computation are used to illustrate how *Implicit Patching* can be used in practice. All the provided examples have been implemented in *Marsyas-0.2* and their

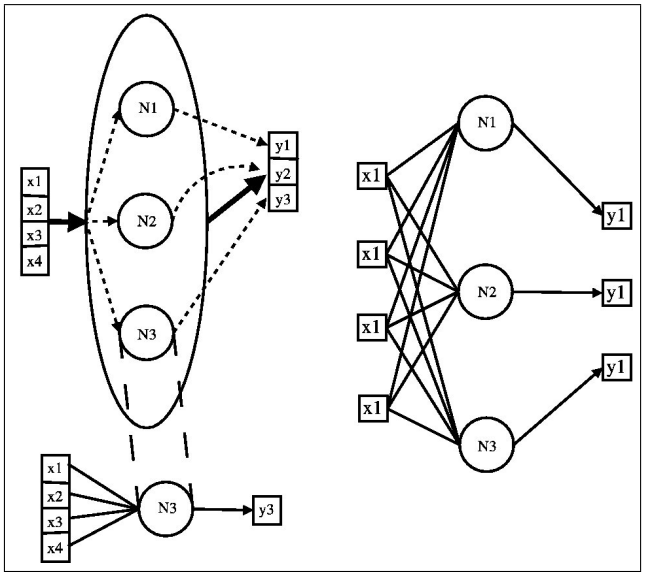

Figure 5. Layer of an Artificial Neural Network

source code is available as part of the distribution.

Figure 6 shows a dataflow network for extracting audio features for real-time music/speech classification. Series connections are top to bottom and Fanout connections are shown by forking horizontally. For example the output of the *Spectrum* calculation is used as input to the *Centroid*, *Rolloff*, and *Flux* MarSystems. The input to texture memory is 4 observations (the features) by 1 sample and the output is 40 samples consisting of the last 40 feature vectors (corresponding to approximately 1 second). Means and variances of the feature vector over the texture window are calculated and the input to the classifier is a 8-dimensional feature vector (4 means, 4 variances). The entire network can be created at run-time without requiring any code compilation. The complete feature extraction front-end described in [21], has been implemented as a dataflow network in *Marsyas-0.2* in a similar fashion.

The Phasevocoder ([20]) is a well-known computationally intensive audio processing technique that allows independent control of pitch and time shifting of sounds. The phasevocoder, in *Marsyas-0.2*, can be fully specified as a dataflow network created with *Implicit Patching*. Even on a Pentium III laptop, this implementation is fast enough to run in real-time. The fact that the entire dataflow network can be specified at runtime together with the use of composites allows interesting applications to be created with minimum effort. For example, it is easy to create a real-time harmonizer, by creating four instances of a phasevocoder network with different amounts of pitch shifting and adding them to a *Fanout* that is connected in *Series* with live audio input. This entire process can be accomplished with minimum effort, no code recompilation while retaining the real-time performance of compiled code. Figure 8 shows how such a harmonizer can be distributed over multiple computers.

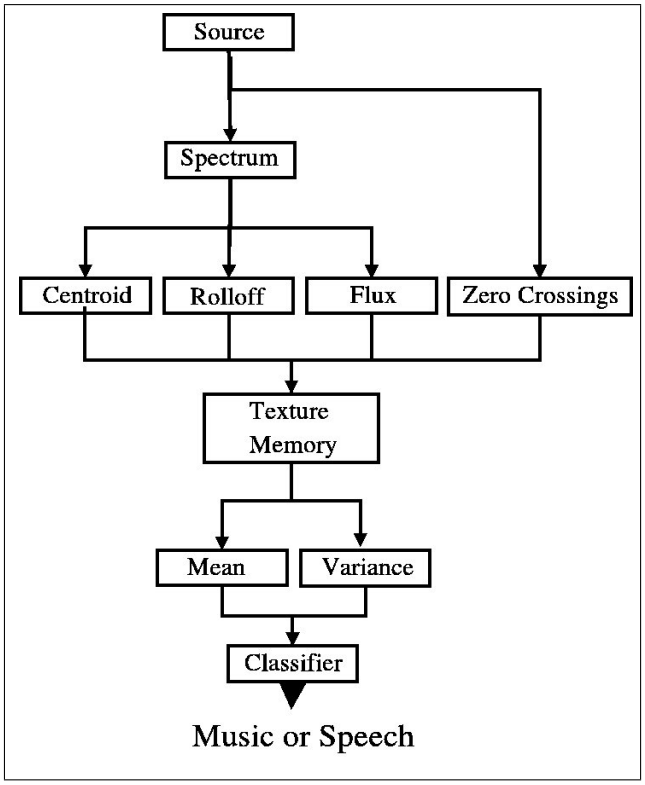

Figure 6. Feature Extraction Network for real-time Music/Speech classification

5.1. Distributing Audio Computation

There are two standard data communication protocols used on the Internet: transmission control protocol (TCP), and user datagram protocol (UDP). They are both part of the TCP/IP protocol suite, and are used in conjunction with the Internet protocol (IP) to provide a bridge for packet delivery to the target application. Packets on an IP network may arrive out of order, damaged, or not at all; hence, TCP is used to manage these issues so that data is received correctly. This extra reliability also means extra overhead, primarily due to acknowledgement messages (ACK), packet retransmissions, flow control, and extra information in the TCP header. The UDP protocol is a much simpler protocol that provides the same bridge to target applications, but no reliability. Commonly referreed to as a best-effort service, UDP therefore is much faster and is the preferred protocol for real-time data such as audio and video. Packets can arrive late, out of order, damaged, or not at all; however, it is up to the application layer to provide a mechanism to deal with these issues.

Marsyas-0.2 supports both the UDP and TCP protocols. In order to send data to another machine, a "network sink" object is simply inserted somewhere in the flow of a *MarSystem*. In order to receive data, a "network source" object is inserted. Control flow and data flow are managed seperately so that controls can be changed from the sender and propagate through the system. The idea is that a user can operate several worker machines and the view of the distributed system is abstracted as one large com-

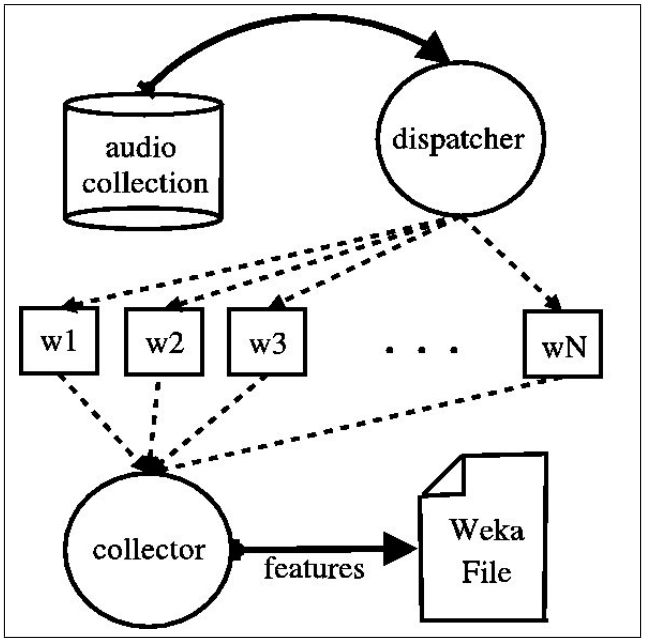

Figure 7. Distributed audio feature extraction

# of Computers	1L	1W	2W	3W
Minutes	9:39	11:48	6:01	5:49

Table 1. Distributed Feature Extraction Experiments (1000 30-second clips)

posite *MarSystem*.

In the next subsections we illustrate the network-aware *MarSystems* by describing two classic applications in audio analysis and synthesis: distributed audio feature extraction, and a real-time distributed harmonizer implemented using multiple instances of a phasevocoder.

5.2. Distributed Audio Feature Extraction

This application uses the TCP protocol in order to guarantee that all data is received and the correct features are calculated. The point of the experiment was to reveal the time advantage of using several machines processing features in parrallel rather than just one machine. A dispatcher process was used to distribute audio files to each machine in the system, and a collector process was used to gather the features as they were calculated (see figure 7. In order to compare the operation to feature calculation on one computer, both the dispatcher and collector were operating on the same machine. Our exrperiment took place on a 100Base-T Ethernet LAN with single processor Apple G5 computers running OS X. We found that using three or four machines gave optimal performance, and adding more machines gave minimal improvement, likely due to the sending capacity of the dispatcher. For these experiments all the data is stored in the dispatcher and there is no replication. Table 1 shows results for extracting feature from a collection of 1000 30-second clips. Computing the features on a single machine is denoted 1-L, and for the other entries the single dispatcher/collector, multiple worker scheme of figure 7 is used. As can be seen, the overhead of network transmission with only 1 worker (1W) doesn't pay off, but after that signficant savings in feature extraction time can be achieved by distributing computation. Of course, if data replication is allowed these results can be further improved.

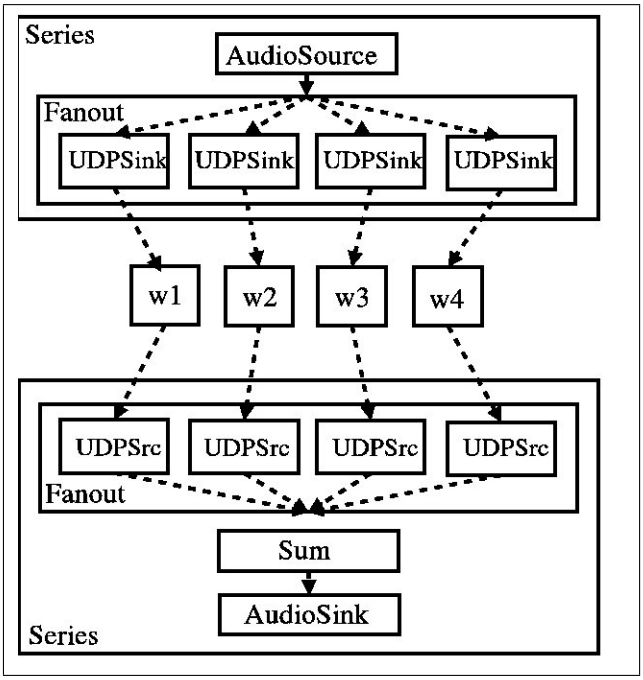

Figure 8. Real-time distributed harmonizer

5.3. Real-time distributed phasevocoder

Transmission of real-time data over packet switched networks has received widespread attention in recent years; most recently with the introduction of the Real-Time Transport Protocol (RTP). This protocol standardizes a way for programs to manage time critical data over networks, providing isochronous data transfer between hosts. Typically managed on top of the UDP protocol in the application layer, RTP provides mechanisms such as timestamps and sequence numbers, so that packets that arrive late or out of sequence can simply be discarded. Marsyas does not yet support the RTP protocol; however, due to negligable propagation delay on a 100Base-T LAN we decided to test the application with UDP only. Future work will indeed consist of providing support for RTP in Marsyas.

The goal of this experiment was to parallelize audio synthesis over several machines and collect their sum at a destination point. Audio was sent from one machine to several worker machines. Each worker machine, pitch shifts the audio differently using a phasevocoder. The results are all sent to one destination where they are summed and played back. Figure 8 is a diagram of this process.

6. DISCUSSION

In this paper, *Implicit Patching* an extension to the usual dataflow model used in Computer music was presented. This model addresses some of the limitations of existing systems by providing flexible audio and control rates, dynamic readjusting of buffer sizes and a structured approach to creating dataflow networks that is based on object composition rather than explicit patching. Examples from audio analysis, synthesis and distributed processing were used to illustrate this approach.

Of course there are limitations of *Implicit Patching*. Everytime sound is propagated through the network all MarSystems must finish computing. Therefore, if somewhere in the network there is a very slow *MarSystem* it will determine the overall speed of processing. More general dataflow environments don't have that limitation at the cost of more complex implementation and performance penalties. Another limitation is the inability to express feedback loops on the dataflow level using the existing Composites. One possible solution we are exploring is to try to come up with Composites for feedback loops.

Finally, even though *Implicit Patching* was presented in contrast to *Explicit Patching* they are not mutually exclusive. For example it is possible to still have multiple input/output ports or explicitely make connections (for example for feedback loops) on top of the dataflow infrastructure described in this paper. It is our hope that that *Implicit Patching* will be useful in other programming languages and environments for audio processing.

7. REFERENCES

[1] W. Ackerman, "Data flow languages," *IEEE Computer*, vol. 15, no. 2, pp. 15–25, 1982.

[2] J. Backus, "Can programming be liberated from the von neumann style ? a functional style and its algebra of programs," *Communications of the ACM*, vol. 21, no. 8, pp. 613–641, Aug 1978.

[3] W. Wadge and E. Ashcroft, *Lucid, the dataflow programming language*, ser. APIC Studies in Data Processing. New York, NY: Academic Press, 1985.

[4] W. Johnston, J. Paul Hanna, and R. Millar, "Advances in dataflow programming languages," *ACM Computing Surveys*, vol. 36, no. 1, pp. 1–34, March 2004.

[5] D.-A. Manolescu, "A data flow pattern language," in *Proceedings of the 4th Pattern Languages of Programming*, Monticello, Illinois, September 1997.

[6] J. Morrison, *Flow-based Programming: A New Approach to Application Development*. New York, NY: van Nostrand Reinhold, 1994.

[7] J. Chadabe, "The voltage-controlled synthesizer," in *The development and practice of electronic music*, J. Appleton, Ed. Prentice-Hall, New Jersey, 1975.

[8] R. Boulanger, *The Csound book*. Cambridge, Mass.: MIT Press, 2000.

[9] D. Zicarelli, "How i learned to love a program that does nothing," *Computer Music Journal*, vol. 26, no. 4, pp. 44–51, 2002.

[10] M. Puckette, "Max at seventeen," *Computer Music Journal*, vol. 26, no. 4, pp. 31–43, 2002.

[11] A. Xavier, "An object-oriented metamodel for digital signal processing with a focus on audio and music," Ph.D. dissertation, Univ. of Pompeu Fabra, 2005.

[12] R. Dannenberg, "Machine tongues xix: Nyquist, a language for composition and osound," *Computer Music Journal*, vol. 21, no. 3, pp. 50–60, 1997.

[13] G. Tzanetakis and P. Cook, "Marsyas: A framework for audio analysis," *Organised Sound*, vol. 4(3), 2000.

[14] P. Cook and G. Scavone, "The Synthesis Toolkit (STK), version 2.1," in *Proc. Int. Computer Music Conf. ICMC*. Beijing, China: ICMA, Oct. 1999.

[15] G. Wang and P. Cook, "Chuck: A concurrent, on-the-fly audio programming language," in *Proc. International Computer Music Conference (ICMC)*, Singapore, September 2003.

[16] R. Dannenberg and E. Brandt, "A flexible real-time software synthesis system," in *Proc. International Computer Music Conference (ICMC)*, 1996, pp. 270–273.

[17] D. Schwarz and Wright.M., "Extensions and applications of the sdif sound description interchange format," in *Proc. International Computer Music Conference (ICMC)*, 2000.

[18] M. Wright and A. Freed, "Open sound control: A new protocol for communicating with sound syntesizers," in *Proc. Int. Computer Music Conference (ICMC)*, Thessaloniki, Greece, 1997, pp. 101–104.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[20] M. Dolson, "The phase vocoder: A tutorial," *Computer Music Journal*, vol. 10, no. 4, 1986.

[21] G. Tzanetakis and P. Cook, "Musical Genre Classification of Audio Signals," *IEEE Trans. on Speech and Audio Processing*, vol. 10, no. 5, July 2002.

[22] R. Dannenberg, "Combining visual and textual representations for flexible interactive audio signal processing," in *Proc. International Computer Music Conference (ICMC)*, 2004.

[23] W. Grosky, R. Jain, and R. Mehrotra, Eds., *The handbook of Multimedia Information Management*. Prentice Hall, 1997.