CHUGENS, CHUBGRAPHS, CHUGINS: 3 TIERS FOR EXTENDING CHUCK

Spencer Salazar     Ge Wang

Center for Computer Research in Music and Acoustics
Stanford University {spencer, ge}@ccrma.stanford.edu

ABSTRACT
The ChucK programming language lacks straightforward mechanisms for extension beyond its built-in unit generators and processing facilities. Chugens address this issue by allowing programmers to live-code new unit generators in ChucK in real-time. Chubgraphs also allow new unit generators to be built in ChucK, by defining specific arrangements of existing unit generators. ChuGins allow a wide array of high-performance unit generators and general functionality to be exposed in ChucK by providing a dynamic binding between ChucK and native C/C++-based compiled code. Performance and code analysis shows that the most suitable approach for extending ChucK is situation-dependent.

1. INTRODUCTION
Since its introduction, the ChucK programming language [14] has become a popular tool for computer music composers, educators, and application software developers. However, to date, its catalogue of audio processing unit generators and general programming functionality has been largely limited to those that are built-in when the ChucK binary executable is compiled. Adding new unit generators mandates recompilation of the entirety of ChucK, requiring a level of expertise and motivation reserved for an elite group of power-users. Furthermore, community-based development efforts are hampered by this centralization of functionality, as developers of new unit generators have no easy way to share their work.

The aim of the work described herein is to provide ChucK with multiple levels of extendibility, each essential and appropriate to specific tasks and levels of user expertise. On the one hand, ChucK’s pervasive ethos of on-the-fly development creates the desire to design and implement new audio processors in ChucK itself in real-time, working down to the per-sample level if necessary. Furthermore, implementing these components in ChucK allows their use on any operating system ChucK supports with no additional effort from the developer. For these cases we have developed Chugens and Chubgraphs.

On the other hand, real-time performance requirements often mandate the use of compiled native machine code for complex audio-rate processing. There also exists a wealth of C/C++-based software libraries for audio synthesis and effects, such as FluidSynth [4] and Faust [9]. These situations can be straightforwardly handled given ported bindings between ChucK and native compiled code, which is precisely the intent of ChuGins.

2. RELATED WORK
Extensibility is a primary concern of music software of all varieties. The popular audio programming environments Max/MSP [15], Pure Data [10], SuperCollider [8], and Csound [4] all provide mechanisms for developing C/C++-based compiled sound processing functions. Max also allows control-rate functionality to be encapsulated in-situ in the form of Javascript code snippets. Max’s Gen facility dynamically compiles audio-rate processors, implemented as either data-flows or textual code, to native machine code. Csound allows the execution of Tcl, Lua, and Python code for control-rate and/or audio-rate manipulation and simulation. The Impromptu environment supports live-coding of audio-rate processors in a Scheme-like language [3], and LuaAV does the same for Lua [13].

Pure allows programmers to dynamically write and execute audio signal processors using Faust [5]. A performance of compiled machine code, written in C++ and compiled to native machine code, executes audio-rate processors, implemented as either data-flows or textual code, to native machine code.

In the case of an audio synthesizer that does not process an existing signal, the input sample may be ignored. A Chugen defined so may be integrated into audio graphs like any standard ChucK ugen. Since the tick function is just a standard ChucK class member function, it can be as simple or as elaborate as required. Standard library calls, file I/O, multiprocessing (using spork), and other general ChucK programming structures can be integrated into the tick function and supporting code.

For performance reasons, it’s important to consider that the tick function will be called for every sample of audio, so simple ticks functions will typically perform better. Moreover, the intrinsic overhead of ChucK’s virtual machine architecture will cause Chugens to underperform compared to a native C/C++ implementation. Lastly, since Chugens are fundamentally a specialization of a Chug class, it may be advantageous to provide structured access to whichever parameters it wishes to expose to the programmer.

3. CHUGENS, CHUBGRAPHS, CHUGINS
3.1. Chugens
The goal of Chugens (pronounced “chyo-jen”) is to facilitate rapid prototyping of audio synthesis and processing algorithms. Additionally, Chugens provide a basic framework for extending ChucK’s built-in audio processing functionality. Using the Chugen system, a programmer can implement audio-rate, audio algorithms within the ChucK development environment, utilizing the full array of programming facilities provided by ChucK. These processing units can be naturally integrated into standard ChucK programs, even in the same script file, providing seamless control of audio-rate processing, control-rate manipulation, and higher-level compositional organizations.

A Chugen is created first by subclassing the built-in Chugen class. This subclass is required to implement a tick function, which accepts a single floating-point argument (the input sample) and returns a single floating-point value (the output sample). For example, this code uses a Chugen to synthesize a sinusoid using the cosine function:

```chuck
class MyCosine extends Chugen
	0 => float p;
	440 => float f;
	second => ramp => float SRATE;
	fun float tick(float in)
		return Math.cos(p++*2*pi*f/SRATE);
}
```

In the case of an audio synthesizer that does not process an existing signal, the input sample may be ignored.

A Chugen defined so may be integrated into audio graphs like any standard ChucK ugen. Since the tick function is just a standard ChucK class member function, it can be as simple or as elaborate as required. Standard library calls, file I/O, multiprocessing (using spork), and other general ChucK programming structures can be integrated into the tick function and supporting code.

For performance reasons, it’s important to consider that the tick function will be called for every sample of audio, so simple ticks functions will typically perform better. Moreover, the intrinsic overhead of ChucK’s virtual machine architecture will cause Chugens to underperform compared to a native C/C++ implementation. Lastly, since Chugens are fundamentally a specialization of a Chug class, it may be advantageous to provide structured access to whichever parameters it wishes to expose to the programmer.

3.2. Chubgraphs
Chubgraphs (pronounced “chub-graph”) provide a way to construct new unit generators by composition, arranging multiple existing ugens into a single unit. In this way, common arrangements of existing unit generators can be defined and instantiated. Furthermore, Chubgraph parameters can be exposed in a structured manner via class member functions.

A Chubgraph is defined by extending the Chug class. The Chug class has member variables named inlet and outlet; inlet is a ugen that represents the input signal to the Chubgraph, and outlet is the output signal. The Chug class’s internal audio processing graph is created by spanning a sequence of ugens between inlet and outlet. The following Chubgraph implements a basic feedback echo processor:

```chuck
class Feedback extends Chug
	inlet => Gain dry => outlet;
	dry => Delay delay => outlet;
	delay => Gain feedback => delay;

0.8 => feedback.gain;
```

(Chugens that do not wish to process an input signal, such as audio synthesizing algorithms, may omit the connection from `inlet`.)

Compared to Chugens, Chubgraphs have obvious performance advantages, as primary audio-rate processing still occurs in the native machine code underlying its component ugens. However Chugens are limited to audio algorithms that can be expressed as combinations of existing unit generators; implementing, for example, intricate mathematical formulas or conditional logic in the form of a ugen graph is possible but is, in our experience, fraught with hazard.

3.3. ChuGins
ChuGins (pronounced “chug-in”) allow near limitless possibilities for expansion of ChucK’s capabilities. A ChuGin is a distributable dynamic library, typically written in C or C++ compiled to native machine code, which ChucK can be instructed to load at runtime. When loaded, the ChuGin defines one or more classes that are subsequently available to ChucK programs. These classes may define new unit generators or provide general programmatically parameterizable functionality beyond that built in to ChucK. Since these classes are normal ChucK classes implemented with native code, member functions and parameters can be used to provide an interface to control parameters.

ChuGins are best suited for audio algorithms that are reasonably well understood and stand to gain from the “write Compile-run” development cycle and C/C+-based programming mandated by ChuGins makes implementing purely audio-rate processors require comparatively more effort than the Chugraph or Chugen approaches. However for ugens a programmer intends to use over an extended period of time, the effort to implement a ChuGin will quickly pay off in the form of lower CPU usage.

An additional advantage of ChuGins is that they may provide functionality far outside the intrinsic capabilities of ChucK. Complex synthetic C/C+-based synthesis packages can be imported wholesale into ChucK, opening up an abundance of sonic possibilities. For example, ChuGins have been implemented to bring audio processing programs from the Faust programming language into ChucK. Similarly, the SoundFont renderer FluidSynth has been implemented as a ChuGin. This functionality is not limited to audio processing; a serial port input/output ChuGin is under development, as are other, currently unimplemented, possibilities.

Development of a ChuGin is somewhat more complex than Chugraphs or Chugens, and does not lend itself to explicit presentation of code herein. Using a set of conventions macro, a ChuGin developer first defines a query function, which ChucK calls upon first loading the ChuGin. ChucK provides the query function with routines with which to define unit generators and classes, specify what member variables and functions...
CHUGENS, CHUGGRAPHS, CHUGINS: 3 TIERS FOR EXTENDING CHUCK

Spencer Salazar  
Ge Wang  
Center for Computer Research in Music and Acoustics  
Stanford University  
{spencer, ge}@ccrma.stanford.edu

ABSTRACT

The ChucK programming language lacks straightforward mechanisms for extension beyond its built-in programming and processing facilities. Chugens address this issue by allowing programmers to live-code new unit generators in ChucK in real-time. Chugens also allow new unit generators to be built in ChucK, by defining specific arrangements of existing unit generators. Chugens allows a wide array of high-performance unit generators and general functionality to be exposed in ChucK by providing a dynamic binding between ChucK and native C/C++-based compiled code. Performance and code analysis shows that the most suitable approach for extending ChucK is situation-dependent.

1. INTRODUCTION

Since its introduction, the ChucK programming language [14] has become a popular tool for computer music composers, educators, and application software developers. However, to date, its catalogue of audio processing unit generators and general programming functionality has been largely limited to those that are built-in when the ChucK binary executable is compiled. Adding new unit generators mandates recompilation of the entirety of ChucK, requiring a level of expertise and motivation reserved for an elite group of power-users. Furthermore, community-based development efforts are hampered by this centralization of functionality, as developers of new unit generators have no easy way to share their work.

The aim of the work described herein is to provide ChucK with multiple levels of extensibility, each essential and appropriate to specific tasks and levels of user expertise. On the one hand, ChucK’s pervasive ethos of on-the-fly development creates the desire to design and implement new audio processors inChucK itself in real-time, working down to the per-sample level if necessary. Furthermore, implementing these components inChucK allows their use on any operating system ChucK supports with no additional effort from the developer. For these cases we have developed Chugens and Chuggraps.

On the other hand, real-time performance requirements often mandate the use of compiled native machine code for complex audio-rate processing. There also exists a wealth of C/C++-based software libraries for audio synthesis and effects, such as FluidSynth [4] and Faust [9]. These situations can be straightforwardly handled given portable bindings between ChucK and native compiled code, which is precisely the intent of ChugIns.

2. RELATED WORK

Extensibility is a primary concern of music software of all varieties. The popular audio programming environments Max/MSP [15], Pure Data [10], SuperCollider [8], and Csound [4] all provide mechanisms for developing C/C++-based compiled sound processing functions. Max also allows control-rate functionality to be encapsulated in-situ in the form of Javascript code snippets. Max’s Gen facility dynamically compiles audio-rate processors, implemented as either data-flows or textual code, to native machine code. Csound also allows the execution of Tcl, Lua, and Python code for control-rate and/or audio-rate manipulation and simulation. The Impromptu environment supports live-coding of audio-rate processors in a Scheme-like language [3], and LuaAV does the same for Lua [13]. Pure allows programmers to dynamically write and execute audio signal processors using Faust [5]. A performance of compiled machine code written in a more portable audio workstation software, in the form of VST, RTAS, AudioUnits, and LADSPA plugins developed primarily in C and C++. For general-purpose computing, JNI provides a highly flexible binding between native machine code and the Java virtual machine-based run-time environment [7]. RubyGems is a complete plugin and package management tool for both Ruby-based plugins and C/C++ libraries compiled with Ruby bindings [11].

3. CHUGENS, CHUGGRAPHS, CHUGINS

3.1. Chugens

The goal of Chugens (pronounced “chyoo-jens”) is to facilitate rapid prototyping of audio synthesis and processing algorithms. Additionally, Chugens provide a basic framework for extending ChucK’s built-in audio processing functionality. Using the Chugen system, a programmer can implement sample-rate audio algorithms within the ChucK development environment, utilizing the full array of programming facilities provided by ChucK. These processing units can be naturally integrated into standard ChucK programs, even in the same script file, providing seamless control of audio-rate processing, control-rate manipulation, and higher-level compositional organization.

Chugen is created first by subclassing the built-in Chugen class. This subclass is required to implement a tick function, which accepts a single floating-point argument (the input sample) and returns a single floating-point value (the output sample). For example, this code uses a Chugen to synchronize a sinusoid using the cosine function:

```chuck
class MyCosine extends Chugen

0 => int p;
440 => float f;
second/samp => float STAPE;

fun float tick(float in)

	return Math.cos(p+2*pi*f/STAPE);
}
```

In the case of an audio synthesizer that does not process an existing signal, the input sample may be ignored. A Chugen defined so may be integrated into audio graphs like any standard ChucK ugen. Since the tick function is just a standard ChucK class member function, it can be as simple or as elaborate as required. Standard library calls, file I/O, multiprocessing (using spawn), and other general ChucK programming structures can be integrated into the tick function and supporting code. For performance reasons, its important to consider that the tick will be called for every sample of audio, so simple ticks functions will typically perform better. Moreover, the intrinsic overhead of ChucK’s virtual machine architecture will cause Chugens to underperform compared to a native C/C++ implementation. Lastly, since Chugens are fundamentally a specialization of a Chug class, it may require modifications to provide structured access to whichever parameters it wishes to expose to the programmer.

3.2. Chuggraps

Chuggraphs (pronounced “chug-graph”) provide a way to construct new unit generators by composition, arranging multiple existing ugens into a single unit. In this way, common arrangements of existing unit generators can be defined and instantiated. Furthermore, Chuggraph parameters can be exposed in a structured manner via class member functions.

A Chuggraph is defined by extending the Chuggraph class. The Chuggraph class has member variables named inlet and outlet; inlet is an ugen that represents the input signal to the Chuggraph, and outlet is the output signal. The Chuggraph’s internal audio processing graph is created by spanning a sequence of ugens between inlet and outlet. The following Chuggraph implements a basic feedback echo processor:

```chuck
class Feedback extends Chuggraph

| inlet => Gain dry => outlet;
dry => Delay delay => outlet;
delay => Gain feedback => delay;
0.8 => feedback.gain;
```

(Chuggraphs that do not wish to process an input signal, such as audio synthesizing algorithms, may omit the connection from inlet.)

Compared to Chugens, Chuggraphs have obvious performance advantages, as primary audio-rate processing still occurs in the native machine code underlying its component ugens. However Chuggraps are limited to audio algorithms that can be expressed as combinations of existing unit generators; implementing, for example, intricate mathematical formulas or conditional logic in the form of ugen graph is possible but in our experience, fraught with hazard.

3.3. ChugIns

ChugIns (pronounced “chug-in”) allow near limitless possibilities for expansion of ChucK’s capabilities. A ChugIn is a distributable dynamic library, typically written in C or C++ compiled to native machine code, which ChucK can be instructed to load at runtime. When loaded, the ChugIn defines one or more classes that are subsequently available to ChucK programs. These classes may define new unit generators or provide general-purpose programming functionality beyond that built in to ChucK. Since these classes are normal ChucK classes implemented with native code, member functions and variables can be used to provide an interface to control parameters.

ChugIns are best suited for audio algorithms that are reasonably well understood and stand to gain from the thronging of compiled machine code. The “write-compile-run” development cycle and C/C++-based programming mandated by ChugIns makes implementing audio algorithms in ChugIns comparatively more effort than the Chuggraph or Chugen approaches. However for ugens a programmer intends to use over an extended period of time, the effort to implement a ChugIn will quickly pay off in the form of lower CPU usage.

An additional advantage of ChugIns is that they may provide functionality far outside the intrinsic capabilities of ChucK. Complex synthesis C/C++-based synthesis packages can be imported wholesale into ChucK, opening up an abundance of sonic possibilities. For example, ChugIns have been implemented to bring audio processing programs from the Faust programming language into ChucK. Similarly, the SoundFont renderer FluidSynth has been implemented as a ChugIn. This functionality is not limited to audio processing; a serial port input/output ChugIn is under development, as are other multimedia processors.

Development of a ChugIn is somewhat more complex than Chuggens or Chuggraphs, and does not lend itself to explicit presentation of code herein. Using a set of convenient macros, a ChugIn developer first defines a query function, which ChucK calls upon first loading the ChugIn. ChucK provides the query function with routines with which to define unit generators and classes, specify what member variables and functions
are associated with these, and indicate a tick function in the case of unit generators. These functions are then defined in C/C++, using predefined macros for interactions with the upper-level Chuck runtime, such as retrieving function arguments, getting and setting member variables, and handling input/output samples. This code is then compiled into a dynamic library using the standard facilities for doing so on the target computing platform (gcc for Mac OS X and Linux systems, Visual C++ for Windows systems). Additional C/C++ code or libraries, such as STK, may be compiled into the dynamic library using the mechanisms standard for those operations on the target platform. Furthermore, Faust DSP code can be automatically converted to ChuGin form using Faust's just architecture file and an accompanying compilation script.

4. PERFORMANCE CASE STUDIES

Execution speed of audio software is typically a vital metric for computer musicians, as real-time audio synthesis requires timely production of tens of thousands of samples per second. To evaluate the performance of our mechanisms for extending Chuck, we designed and implemented several reference unit generators using our mechanisms for extending ChucK, we designed and implemented several reference unit generators using each method. For each method, we measured the time required to offline-render 5 minutes of audio using the resulting unit generator.

In many situations not only is CPU time at a premium, but a programmer's time to implement a specified application is also limited. Therefore, often “less complex” programs are more desirable than “more complex” programs because they can be developed faster and be better understood by other programmers. While quantifying code complexity is a nuanced and complex task of finding and installing ChucK extensions.

4.1. ChuGin

For this case study we implemented a simple comb filter, the “plucked string filter” described in [12]. For reference, the comparatively concise ChuGin version is as follows:

```java
public class CombiFilter extends CombiGenerator { 
    float in = 0.0f; 
    float out = 0.0f; 
    float delta = 0.0f; 
    float amp = 0.0f; 
    float freq = 0.0f; 
    public CombiFilter() { } 
    public void processSample(float in) { 
        float out = in - 0.0f;
        out = 0.0f; 
        return out; 
    } 
} 
```

Table 1. Performance and complexity measurements for CombiFilter.

<table>
<thead>
<tr>
<th>CombiFilter</th>
<th>Time (s)</th>
<th>Lines of Code</th>
<th>Aptness</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChuGin</td>
<td>10.944</td>
<td>18</td>
<td>197</td>
</tr>
<tr>
<td>Chugraph</td>
<td>2.841</td>
<td>9</td>
<td>26</td>
</tr>
<tr>
<td>ChuGin</td>
<td>2.001</td>
<td>65</td>
<td>130</td>
</tr>
</tbody>
</table>

Table 2. Performance and complexity measurements for Bitcrusher.

<table>
<thead>
<tr>
<th>Bitcrusher</th>
<th>Time (s)</th>
<th>Lines of Code</th>
<th>Aptness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chugen</td>
<td>10.600</td>
<td>21</td>
<td>223</td>
</tr>
<tr>
<td>Chugraph</td>
<td>6.544</td>
<td>41</td>
<td>268</td>
</tr>
<tr>
<td>ChuGin</td>
<td>1.965</td>
<td>84</td>
<td>165</td>
</tr>
</tbody>
</table>

5. FUTURE WORK

Further developments we are pursuing include mechanisms for creating general-purpose programming libraries in ChucK. Additionally we are investigating a unified ChuGin repository and distribution system, similar to Debian Apt [1] or RubyGems, to simplify the task of finding and installing ChucK extensions.

As can be seen in Table 1, the ChuGin variant of CombiFilter performs best, but the ChuGraph version is not far behind. The ChuGraph version performs poorly as a result of requiring execution of Chuck code at audio-rate, and also requires more code than the ChuGraph, as the delay line and moving average filter needed to be implemented from scratch. The vastly superior lines of code and aptness metrics, the ChuGraph is probably the best approach to use in this case. ChuGraph excels here because CombiFilter is essentially a particular arrangement of existing unit generator primitives, i.e. Delay and OneZero, allowing a concise and efficient implementation. A ChuGin may be more appropriate if speed is the utmost concern, but its inferior code complexity makes it largely undesirable in this class of audio processing algorithm.

Chugens generate audio by executing ChuGin virtual machine code at audio-rate; dynamically compiling ChucK code.

Additional improvements may be required in the form of process and memory safety in ChuGins. Currently, ChuGins execute in the same process space as Chuck itself, which means that buggy ChuGins can crash Chuck outright. Technological approaches can alleviate these problems somewhat, but organizational solutions may also be desirable, such as having registries of vetted and thoroughly tested ChuGins.

7. REFERENCES


are associated with these, and indicate a tick function in the case of unit generators. These functions are then defined in C/C++, using predefined macros for interactions with the upper-level Chuck runtime, such as retrieving function arguments, getting and setting member variables, and handling input/output samples.

This code is then compiled into a dynamic library using the standard mechanisms for doing so on the target computing platform (gcc for Mac OS X and Linux systems, Visual C++ for Windows systems). Additional C/C++ code or libraries, such as STK, may be compiled into the dynamic library using the mechanisms standard for those operations on the target platform. Furthermore, Faust DSP code can be automatically converted to Chuck form using Faust's autophrase file and an accompanying compilation script.

4. PERFORMANCE CASE STUDIES

Execution speed of audio software is typically a vital metric for computer musicians, as real-time audio synthesis requires timely production of tens of thousands of samples per second. To evaluate the performance of our mechanisms for extending Chuck, we designed and implemented several reference unit generators using each method. For each method, we measured the time required to offline-render 5 minutes of audio using the resulting unit generator.

In many situations not only is CPU time at a premium, but a programmer's time to implement a specified audio solution is of utmost importance. Therefore, often “less complex” programs are more desirable than “more complex” programs because they can be developed faster and be better understood by other programmers. While quantifying complexity code is a nuanced and variable field of inquiry, we have chosen to represent it in the form of lines of code required to implement the unit generator using each method. In addition to characterizing code complexity, lines of code gives an approximation of much time the extension may have taken to develop.

To construct an overall picture of performance vs. development effort, we have created a composite statistic formed by the product of these two metrics, named “aptness.” A lower aptness indicates that a particular technique is more appropriate for a particular processing application than a technique with a higher aptness.

4.1. CombFilter

For this case study we implemented a simple comb filter, the “plucked string filter” described in [12]. For reference, the comparatively concise Chuck version is as follows:

```chuck
public class CombFilter extends Chubgraph {
    inlet => Delay d => outlet;
    d => OneZero os => d;  // <1 os.szero;
    100.0 => ramp => d.delay;
    0.75 => os.gain;
}
```

As can be seen in Table 1, the ChuGin variant of CombFilter performs best, but the Chuck version is not far behind. The Chugen version performs poorly as a result of requiring execution of Chuck code at audio-rate, and also requires more code than the ChuckFilter as the delay line and moving average filter needed to be implemented from scratch. In contrast to the vastly superior lines of code and aptness metrics, the ChuckFilter is probably the best approach to use in this case. ChuckFilter excels here because CombFilter is essentially a particular arrangement of existing unit generator primitives, i.e. Delay and OneZero, allowing a concise and efficient implementation. A ChuGin may be more appropriate if speed is the utmost concern, but its inferior code complexity makes it largely undesirable in this class of audio processing algorithms.

5. FUTURE WORK

Further developments we are pursuing include mechanisms for creating general-purpose programming libraries in Chuck. Additionally we are investigating a unified ChuGin repository and distribution system, similar to Debian APT [1] or RubyGems, to simplify the task of finding and installing Chuck extensions.

Our current system of importing ChuGins does not scale well, as ChuGins cannot be loaded on demand. Rather, Chuck will load every ChuGin that is installed on a system, which may take a noticeable amount of time if there are many ChuGins present. In the future we wish to more intelligently load ChuGins only when their component ugens and classes are invoked by active Chuck code. ChuGins generate audio by executing Chuck virtual machine code at audio-rate; dynamically compiling ChuGins instead to native machine code may introduce significant performance gains. Environments such as LuaAV, Gen, and Pure do just this, compiling high-level programs to native code for audio-rate execution.

Additional improvements may be required in the form of process and memory safety in ChuGins. Currently, ChuGins execute in the same process space as Chuck itself, which means that buggy ChuGins can crash Chuck outright. Technological approaches can alleviate these problems somewhat, but organizational solutions may also be desirable, such as having registries of vetted and thoroughly tested ChuckGins.

6. ACKNOWLEDGEMENTS

Special thanks to Kassen Oud, Casper Schipper, Jorge Herrera, and Hongchao Shi for their invaluable beta testing, bug finding, and feature requesting. Additional support for this research was made possible by a National Science Foundation Creative IT Grant, No. NS-0855758.

7. REFERENCES


