FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Supernova - A Multiprocessor Aware Real-Time Audio Synthesis Engine For SuperCollider

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Musikinformatik

eingereicht von

Tim Blechmann
Matrikelnummer 0526789

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Ao. Univ. Prof. Dr. M. Anton Ertl

Wien, 09.06.2011

(Unterschrift Verfasser/in)

(Unterschrift Betreuer/in)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Supernova - A Multiprocessor Aware Real-Time Audio Synthesis Engine For SuperCollider

Master Thesis

In partial fulfillment of the

Master degree

in

Music Informatics

Presented by

Tim Blechmann
Matriculation Number 0526789

at the
Faculty of Informatics, Vienna University of Technology

Supervisor: Ao. Univ. Prof. Dr. M. Anton Ertl

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 09.06.2011

(Unterschrift Verfasser/in)

(Unterschrift Betreuer/in)

Abstract

These days, most computer systems are built with multi-core processors. However, most computer music systems use a single-threaded synthesis engine. While the single-core performance for current CPUs is sufficient for many computer music applications, some demanding applications benefit from using multiple processors simultaneously.

Real-time audio synthesis imposes specific constraints regarding real-time safety, since the synthesis engine needs to be tuned for worst-case latencies below one millisecond. As a result, no blocking synchronization can be used and the signal processing graph needs to be split into reasonably sized parts, that provide enough parallelism without introducing a significant scheduling overhead.

During the work on this master thesis, I developed **Supernova** as a multiprocessor aware synthesis engine for SuperCollider. SuperCollider is a computer music system based on a dynamic scripting language with a real-time garbage collector, that is used to control a separate audio synthesis server. Supernova replaces the original audio synthesis engine. It is not possible to automatically parallelize the synthesis graph of SuperCollider without fundamentally changing the semantics of the SuperCollider class library. Therefore a the programming model for the synthesis graph was extended, exposing parallelism explicitly to the user. To achieve this, I propose two simple concepts, ‘parallel groups’ and ‘satellite nodes’.

To my knowledge, Supernova is the first parallel audio synthesis engine that is designed for real-time operations under low-latency constraints without adding any additional latency to the audio signal.

Kurzfassung

In den meiŝten derzeit gebauten Computern werden Mehrkernprozessoren verbaut. Allerdings benutzen die meissten Computermusik Systeme einen einzigen Thread für die Klangsynthese. Für viele Anwendungsbereiche der Computermusik ist die Rechenleistung eines einzelnen Kerns einer aktuellen CPU ausreichend, allerdings gibt es einige rechenintensive Anwendungsfälle, in denen man von der Rechenleistung mehrerer Kerne Gebrauch machen kann.

Klangsynthese in Echtzeit hat bestimmte Anforderungen bezüglich ihrer Echtzeitfähigkeit, da die Synthese-Engine für worst-case Latenzen von unter einer millisekunde ausgelegt sein muss. Aus diesem Grund dürfen keine blockierenden Synchronisationsstrukturen eingesetzt werden und der Signalverarbeitungsgraph muss in sinnvolle Teile zerlegt werden, die eine ausreichende Parallelität aufweisen ohne einen erheblichen Synchronisationsoverhead zu verursachen.

Während der Arbeit an dieser Masterarbeit habe ich **Supernova** entwickelt, einen mehrprozessorfähiger Klangsynthese Server für SuperCollider. SuperCollider ist ein Computermusik System, das auf einer dynamischen Skriptingsprache mit einem echtzeitfähigen Garbage Collector basiert, mit der ein separater Klangsynthese Server gesteuert wird. Supernova ersetzt den ursprünglichen Klangsynthese Server. Es nicht möglich ist, den Synthesegraphen von SuperCollider automatisch zu parallelisieren, ohne die Semantik der Klassenbibliothek grundlegend zu verändern. Daher wurde das Programmiermodell des Synthesegraphen erweitert, um dem Benutzer eine Möglichkeit zu geben, Parallelismus explizit zu formulieren. Dafür schlage ich zwei einfache Konzepte vor, 'parallele Gruppen' und 'Satellitenknoten'.

Meines Wissens ist Supernova die erste parallele Klangsynthese-Engine, die für Echtzeitanwendungen bei niedrigen Latenzzeiten ausgelegt ist, ohne das Audiosignal zu verzögern.

Contents

Acknowledgments

I would like to thank a few people:

James McCartney for creating and open sourcing SuperCollider

Dr. Dan Stowell for his valuable feedback

Ao. Univ. Prof. Dr. M. Anton Ertl for his guidance when writing this thesis

my parents for supporting me during the whole time of my studies

Chapter 1

Introduction

For many years the number of transistors per CPU has increased exponentially, roughly doubling every 18 months to 2 years. This behavior is usually referred to as ‘Moore’s Law’. Since the early 2000s, this trend does not translate to an increase of CPU performance any more, since the techniques that caused these performance gains have been maxed out [OH05]. Processors have been pipelined, executing single instructions in a sequence of stages, which increases the throughput at the cost of instruction latency. Since the individual stages require a reduced amount of logic, pipelined CPUs allow higher clock rates. Superscalar processors try to dynamically execute independent instructions in parallel. Increasing the CPU frequency would cause an exponential growth in power consumption, imposing practical problems for cooling, mobile applications and, for the musician, fan noise. While power consumption decreases as the die shrinks, it has been suggested to reach its physical limits in the near future [Kis02]. So instead of trying to increase the CPU performance, the computer industry started to increase the number of cores per CPU. These days, most mobile solutions use dual-core processors, while workstations are available with 4 or even up to 8 cores.

The computer music systems that are commonly used these days are designed for single-processor machines [Wes08]. While the single-processor performance of current CPUs is sufficient for many use cases, the limits can quickly be reached when using more complex synthesis algorithms.

1.1 Computer Music Systems

Computer music systems in general consist of two parts: the audio synthesis engine and the control engine, controlling the synthesis. This differentiation was formulated by Max Matthews in the early days of computer music. In his 1969 publication *The Technology of Computer Music* [MMM+69] he wrote:

> “The two fundamental problems in sound synthesis are (1) the vast amount of data needed to specify a pressure function — hence the necessity of a very fast program — and (2) the need for a simple, powerful language in which to describe a complex sequence of sounds.”

Both parts can be used independently. Historically, the first composers using computers were trying to generate structures, e.g. Iannis Xenakis computed velocity trajectories for glissandi for his 1954 orchestral composition “Metastasis” [Dun92] and produced data based on densities and probabilistic distributions for his stochastic compositions [Xen01]. Lejaren Hillier wrote a number of algorithmic compositions where the whole score is generated by computers, starting with his 1957 string quartet “Illiac Suite” [Sup97].

The earliest experiments of digital audio synthesis were carried out by Max Matthews, working as a research engineer at Bell Laboratories [Man04]. In 1957 and 1958, he developed the experimental programs “Music I” and “Music II”, that were quite limited in terms of audio synthesis. In the early 1960s, Matthews extended the program in collaboration with Joan Miller to “Music III” and “Music IV”, which started to interest composers like James Tenney or John Pierce, who realized the first tape compositions with digitally synthesized sounds as Bell Labs. While “Music I” to “Music IV” where written in assembler for specific hardware (IBM 704 and 7094), Matthews developed “Music V” in 1968 using the Fortran programming language.

1.2 SuperCollider & Concepts of Computer Music Systems

SuperCollider can be seen as a distant descendant from the MusicN program family. It was originally developed by James McCartney for MacOS9 on the PPC platform as proprietary software [McC96]. In 2002, SuperCollider 3 was released as open source software under the GNU General Public License, introducing the current client/server architecture. It was then ported to Linux by Stefan Kersten.

It is a very modular system, based on an object-oriented dynamic programming language which is inspired by SmallTalk. The main design decision is a strong separation of synthesis engine and control language. The audio synthesis engine **scsynth** is designed as server [McC02], providing a simple API for controlling the audio synthesis. Scsynth supports both real-time synthesis, playing through an audio interface and non real-time synthesis for generating audio files. The synthesis control language **sclang** was developed as domain-specific programming language for real-time audio synthesis and algorithmic composition, implementing the Johnston-Wilson real-time garbage collector [WJ93]. Sclang provides a huge class library for different kinds of aspects like real-time control of the server, MIDI[1] communication, computer-assisted composition, pattern sequencing or score generation.

Language and Server communicate via an interface of about 70 commands, using a subset of the Open Sound Control (OSC) protocol [WF97]. OSC is a binary protocol that was designed as successor of MIDI, and that is usually transferred via network sockets (usually UDP). Scsynth implements a variation of OSC[2] exposing its interface to external programs via TCP or UDP sockets for real-time operation. For non real-time mode, it can read time-tagged OSC commands from a binary file, that can easily be generated from sclang.

There are several IDEs that support sclang. The first IDE was a native Mac OSX application, that is shipped with the SuperCollider sources. As part of the Linux port, an emacs-based IDE was developed, that is probably the most commonly used interface to sclang beside the OSX app. There are other editor modes for vim, gedit and eclipse, and a python-based IDE, that was originally written when SuperCollider was ported to Windows.

1.2.1 Digital Audio

Digital audio signals are usually represented as sequences of single-precision floating point numbers, although listening tests suggest an improvement in audio quality by using double-precision floating point numbers instead [Gog]. Audio streams are usually processed as **blocks** of samples, where the block size is typically between 64 and 2048 samples. Bigger block sizes typically result in a reduced scheduling overhead and better performance, while increasing the audio latency.

[1] Musical Instrument Digital Interface, industry-standard protocol for communication between audio hardware.

[2] It does not implement address pattern matching and adds an extension for integer addresses.

```
SynthDef(\sweep, {
    arg duration = 10, amp = 0.01, freq_base = 1000, freq_target = 1100;

    var freq = Line.kr(freq_base, freq_target, duration);
    var sine = SinOsc.ar(freq);
    var env  = EnvGen.kr(Env.linen(0.1, duration - 0.5, 0.4, amp),
        doneAction: 2);

    Out.ar(0, sine * env);
}).add;
```

Listing 1.1: SynthDef for a sweeping sine wave

In real-time computer music systems, the engine is triggered by the audio callback[3], which is driven by the interrupt of the audio hardware. During each callback, the backend delivers audio data from the input ports of the device and requests new data to be sent to the audio output.

Audio devices use a small number of blocks to transfer the audio stream between computer and audio hardware. Increasing the number of blocks also increases the audio latency, but can hide missed deadlines e.g. in cases of short CPU peaks. Currently most digital audio hardware supports sampling rates of 44100 Hz[4] and 48000 Hz. In addition to that, most professional hardware supports sampling rates of 96 kHz or even 192 kHz. Running a system with 64 samples per block at 48 kHz, the system needs to schedule one audio interrupt/callback every 1.3 ms, each having a deadline of 1.3 ms. Additional buffering in hardware and digital/audio conversion may increase the latency further than the theoretical minimum of 2 audio buffers for playback.

For computer-based instruments a playback latency below 20 ms is desired [Mag06], for live-processing of percussion instruments round-trip latencies for below 10 ms is required to ensure that original and processed sounds are perceived as single event [Bou00]. So hardware block sizes of down to 64 samples can be necessary.

1.2.2 Organization of Audio Synthesis in SuperCollider

In SuperCollider, instrument definitions are called **synthdefs**. They are built with the function syntax of sclang and are instances of the `SynthDef` class. Listing 1.1 shows a synthdef for a sweeping sine wave. The function syntax is used to build a signal flow graph. A synthdef can be seen as directed acyclic graph (DAG) with ugen as nodes. The resulting ugen graph is shown in Figure 1.1.

Unit Generators

The concept of the ugen is fundamental to most, if not all computer music systems. Unit generators are the smallest atomic units of the synthesis graph, like oscillators, filters, simple arithmetic operators, but also more complex objects like for granular synthesis or machine listening. Unit generators can have parameters, for example a sine wave oscillator will have a frequency parameter, a digital filter parameters for frequency and filter quality. Parameters can be constant numbers, but also the output of other unit generators. Some unit generators also access shared resources like audio busses or buffers.

[3] This applies to backends implementing a ‘pull’ API, that is used by most professional audio driver APIs like Jack, ASIO or CoreAudio

[4] 44100 Hz is the sampling rate of audio CDs

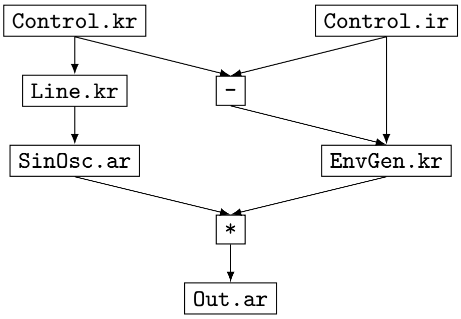

Figure 1.1: Ugen Graph for Listing 1.1

In SuperCollider, Unit Generators are designed as plugins for scsynth which are dynamically loaded at startup. The interface is based on a simple C API. All function calls from the plugin to scsynth are done indirectly using a table of function pointers. The simplicity and clearness of the API makes it easy to load unit generators to other systems. Ugens also have a representation as sclang classes. These classes are usually derived from the abstract `UGen` class.

While typically unit generators are specified explicitly, some ugens are generated implicitly, such as `Control` ugens that are auto-generated from the arguments of the synthdef function, or from operators that are applied on signals. The unit generators for the `sweep` synthdef are listed in Table 1.1.

Signal Rates

Unit generators can be evaluated at different rates, the most commonly used rates are **audio rate**, **control rate** and **instrument rate** signals. Audio rate signals are computed at the sampling rate of the audio backend, usually the audio hardware of the system. Some signals are not supposed to generate audible signals, but to control parameters of the actual audio synthesis. These signals can be computed at a lower rate in order to save CPU time. Typical applications for control-rate signals would be envelopes or LFOs[5]. The control rate of scsynth is specified as a down-sampling factor relative to the audio rate, typically 64. For some applications, higher or lower down-sampling factors are used. Instrument rate signals are computed only once per instrument at the time of instrument creation.

In the SuperCollider language, ugens are usually instantiated explicitly by calling one of the class methods `ar`, `kr` and `ir` of the ugen class to run at audio, control or instrument rate signals. In addition to that, SuperCollider has a fourth notion, called **demand rate**. Demand rate unit generators are not evaluated regularly, but on demand, whenever a special `Demand` unit generator is triggered[6] [M$^+$96]. Table 1.1 also mentions the signal rate of the unit generators from Listing 1.1.

Synthdefs & Synths

Since single unit generators are usually very small, possibly only taking a few hundreds or thousands of CPU cycles, it would result in a high overhead if a graph representation of the ugen graph would have to be traversed for computing a sample block. In order to obtain a

[5] Low Frequency Oscillator, oscillator with a low (i.e. not audible) frequency, commonly used for modulating control values. In contrast to audio oscillators, aliasing effects matter less.

[6] A trigger is defined as transition from a non-positive to a positive value.

Ugen Type	Rate	Function
Control	instrument rate	synth parameters (from function arguments)
Control	control rate	synth parameters (from function arguments)
Line	control rate	frequency ramp
SinOsc	audio rate	generate sine wave signal
-	control rate	computation of the second argument for Env.linen
EnvGen	control rate	envelope
*	audio rate	apply the envelope to the sine wave
Out	audio rate	write the signal to a summing bus

Table 1.1: Unit Generators for a sine sweep

```
// routine to manually schedule synths
~r = Routine({
    Synth(\sweep, [\duration, 10,
        \amp, -12.dbamp,
        \freq_base, 1000,
        \freq_target, 1100]);

    1.0.wait;
    Synth(\sweep, [\duration, 9,
        \amp, -9.dbamp,
        \freq_base, 1000,
        \freq_target, 200]);

    1.5.wait;
    Synth(\sweep, [\duration, 7.5,
        \amp, -18.dbamp,
        \freq_base, 1000,
        \freq_target, 10000]);
});

~r.play // run the routine
```

Listing 1.2: Schedule 3 sweeping sine waves from a routine

higher performance, the ugen graph is topologically sorted, and the unit generators are stored in a sequential data structure, which can be iterated efficiently [Puc91b], reducing the ugen scheduling overhead to a minimum.

The SuperCollider language either sends the linear representation of the synthdefs to the server via OSC or stores it on the file system, from where they can be loaded by the server later. On the server, synthdefs can be instantiated as **synths**. Listing 1.2 shows an example, how 3 different sine sweeps can be scheduled from sclang via a routine. The synths are instantiations of synthdefs on server. In the example, a routine (a rough equivalent to a thread) is started, that schedules notes by creating instances of the Synth class and waiting for a number of seconds. Alternatively SuperCollider provides a pattern sequencer library, that provides a compact syntax to algorithmically generate patterns, that can be used for generating synths on the server. This library also helps a lot when doing non real-time synthesis to generate sound files instead of sending the output to the audio interface in real-time. Listing 1.3 shows the equivalent code

```
// pattern
~p = Pbind(\instrument, \sweep,
    \duration, Pseq([10, 9, 7.5]),
    \db, Pseq([-12, -9, -18]),
    \freq_base, 1000,
    \freq_target, Pseq([1100, 200, 10000]),
    \dur, Pseq([1, 1.5, 0])
);

// play back in real-time
~p.play;

// render soundfile
~p.asScore(10).recordNRT("sweep.osc", "sweep.wav")
```

Listing 1.3: Schedule 3 sweeping sine waves from a pattern

using the pattern library for both real-time and non real-time synthesis.

The SuperCollider Node Graph

When synths are created on the SuperCollider server, they are added at a certain position in the node graph. SuperCollider has the notion of **groups**, which are linked lists of nodes, where each node member can be either a synth or another group, effectively modeling a tree data structure with a group as its root. The position can be specified with one reference node and one relational argument, which can be ‘before’ or ‘after’ a reference node or at ‘head’ or ‘tail’ of a reference group. A more detailed description of the SuperCollider node graph is given in Section 3.1.

1.3 Contribution

To my knowledge, the Supernova synthesis server makes SuperCollider the first general purpose computer music system with multiprocessor support, that is truly dynamic, scalable and can be used for low-latency applications. In order to achieve this, two problems had to be solved.

1. An extension to the SuperCollider node graph was introduced to expose parallelism to the user.

2. A real-time safe audio synthesis engine has been designed, that can meet the latency requirements for computer music.

1.4 Overview

This thesis is divided into the following parts. Chapter 2 analyzes how audio synthesis engines can make use of parallelism. Both data-level parallelism and different types of thread-level parallelism are discussed. In Chapter 3 the SuperCollider node graph is explained in detail, possibilities for automatic parallelization are discussed and two concepts are proposed which would expose parallelism explicitly to the user. Chapter 4 analyses the technical issues when implementing a real-time audio synthesis engine that is suited for low-latency applications, and describes the interesting parts of the implementation of Supernova. It also contains a section on the Nova-SIMD framework, a generic framework for making use of data-level parallelism with a focus on audio synthesis. Chapter 5 shows some experimental results and discusses both throughput and worst-case latency and analyzes the latency hotspots. In Chapter 6 the approach to parallelism of other computer music systems is evaluated according to the definitions that have been discussed in Chapter 2.

Chapter 2

Parallelism in Computer Music Systems

Some researchers like Roger Dannenberg suggest that computer music applications can be characterized as 'embarrassingly parallel' [Dan08]. While this may certainly be true for some applications, it is not necessarily the case in general, since some use cases have a high parallelism, but require a certain amount of synchronization. For our analysis it may be helpful to define a number of concepts that are used in different systems.

Voices

Polyphonic music denotes music, which is using two or more (melodic) voices, each being independent from the others.

Melodies

A melody can be considered as a series of 'notes', unless notes should be played as 'legato'[1], they can be computed independent from each other. The example with 3 sine sweeps from the introduction would fall into this category.

Tracks and Buses

Digital Audio Workstations (DAWs) have the notion of tracks and busses. A track is an entity consisting of an audio input, a soundfile recorder and playback unit, a mixer strip where effects can be added to the audio signal and an audio output. A bus has a similar concept, although it omits the audio playback/recording feature. Figure 2.1 shows a typical DAW. Two stereo tracks ('Tim 1' and 'Tim 2') are mixed down to one stereo bus ('Mix Tim'), which is mixed with the mono track ('Manuel 1') to the 'master' bus.

Channels

Many use cases deal with separate audio signals, each treated as one entity. They are usually referred to as channels. Tracks, busses, samples, audio files or instruments can have multiple channels. The most commonly used stereo signals have two channels. For some applications like ambisonics or wave field synthesis, a large number of channels (dozens to hundreds) is required.

Additive Synthesis

In order to create sounds with specific spectra, additive synthesis is the easiest synthesis technique. A sound is generated by adding the signals of multiple oscillators to build the desired spectrum. Since each oscillator is working as independent unit, parallelizing additive synthesis is very easy in theory, but difficult in practice, since the expenses for each

[1] Notes are played without interruption

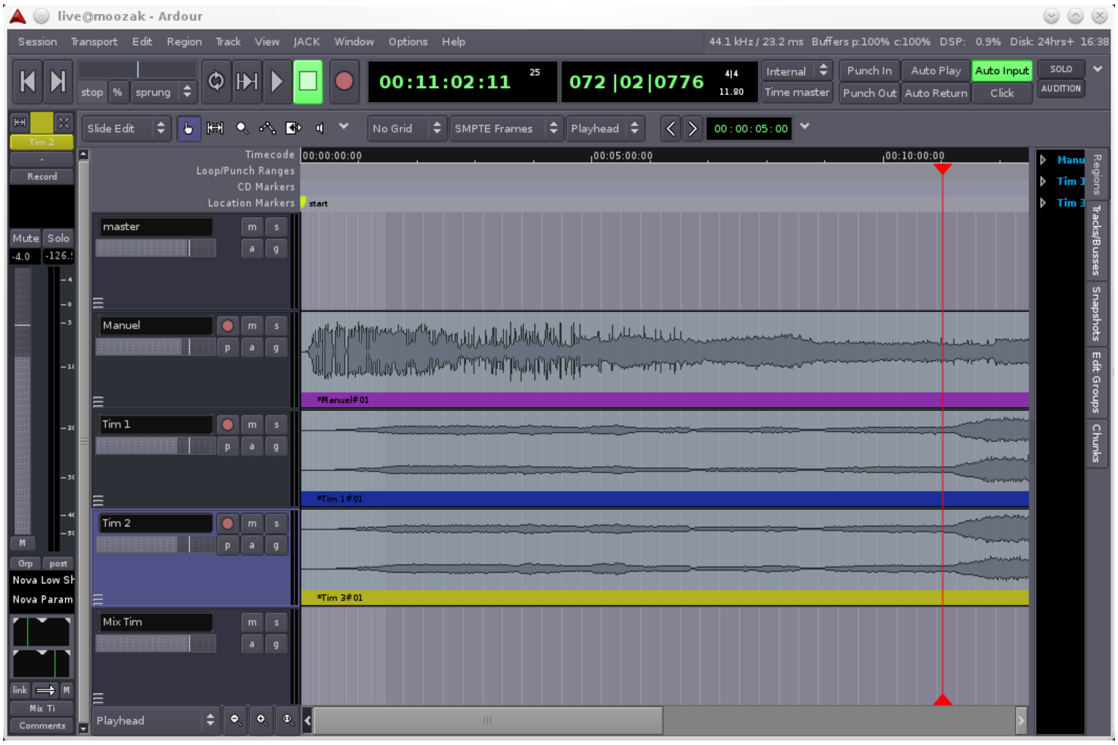

Figure 2.1: A typical DAW session view with 3 tracks and 2 busses

oscillator are rather small, so scheduling overhead would be quite significant (see Section 2.2.2).

Some concepts like channels are clearly embarrassingly parallel, if all channels can be computed completely independently. With tracks, busses or voices it depends on the use case. In some applications they can be computed independently, in others, e.g. if a multi-track recording is mixed down to 2 channels during mastering, or multiple voices should be played back through the same speakers, they require some synchronization. Melodies of independent but possibly overlapping notes, or additive synthesis would require a large synchronization effort.

Apart from making use of data-level parallelism via SIMD instructions, there are two approaches for using thread-level parallelism in computer music systems: one approach is using graph parallelism, the other one is pipelining.

2.1 Data-Level Parallelism: Single Instruction Multiple Data

Today's superscalar CPUs use instruction level parallelism to increase throughput. In particular, SIMD instructions are highly usable for implementing ugen functions. SIMD instructions are special CPU instructions that can process small arrays of data instead of single scalars. E.g. the Intel SSE instruction set family provides instructions for integer and floating point numbers of different size, working on a set of 8 or 16 128-bit registers. Typically instructions are available for element-wise arithmetic, type conversions or bit-wise operations.

2.1.1 Compiler Vectorization

Under certain circumstances, compilers are able to generate SIMD instructions automatically [Nai04]. This optimization is related to loop unrolling and requires some knowledge about the code in order to deal with **aliasing** and **alignment** issues. Aliasing issues occur if two pointers point to overlapping memory regions. If the compiler cannot ensure that arguments don't alias, it cannot generate SIMD instructions, although the program logic may permit it. Similarly, most SIMD instructions require a certain memory alignment that the compiler cannot guarantee. SIMD instructions for non-aligned memory regions may exist, but require extra (and possibly slower) load/store instructions when transfering data between registers and memory. Compilers also generate extra code for loop peeling and versioning, which could be avoided if the specific code is only called under valid alignment and aliasing constraints.

2.1.2 Vectorized Audio Signal Processing

There are several types of ugen functions that benefit from SIMD instructions. The most obvious functions are vector functions like basic arithmetic between signals and scalars, but also signal comparison, clipping, bitmasking (like sign bit clearing or extraction), rounding, mixing, cross fading, panning or waveshaping. Some algorithms, like finding the peak of a signal block need some adaptation, but can also be vectorized. With some vectorization techniques, even branching algorithms can be implemented with SIMD instructions efficiently. Helmut Dersch used branch elimination techniques to implement SIMDfied vector versions of single-precision floating point libm math functions for SSE2, PowerPC and Cell architectures [Der08]. Some math functions are implemented as branching algorithms with different approximation schemes for different parameter intervals. In order to implement a vectorized version of these algorithms, all approximations have to be computed and the actual result needs to be selected by using bitmasks. Although more data is computed than actually needed, the code still significantly outperforms the libm implementations, because the code can be pipelined very well.

Multi-Channel Audio Signal Processing with SIMD

Less common applications for SIMD instructions are multi-channel applications. In contrast to vector or vectorizable code, recursive functions like IIR filters could be computed for multiple channels at the same time. This is especially useful if the number of channels equals the number of samples that can be processed during one SIMD instruction, and if the memory representation of the samples is interleaved. If not, additional instructions are required for packing the argument data to the SIMD register or unpacking the result.

On some SIMD implementations, like the SSE implementation for current Intel processors, the simdfied version of an instruction has the same cost as its single-data equivalent [Inta]. Unfortunately, I am not aware of any implementation that actually makes use of SIMD instructions for multi-channel audio processing. While it is possibly interesting to provide some implementations, it is beyond the scope of this thesis.

2.2 Thread Level Parallelism: Signal Graph Parallelism

As described earlier, there are two notions of signal graphs in SuperCollider, ugen graphs and node graphs. While both types share a similar structure (both are directed acyclic graphs) and can therefore be parallelized with the same algorithms, different CPU expenses for ugens and nodes imply different strategies for parallelization. The ugen graph parallelism can be considered as **fine-grained**, while the node graph parallelism is **coarse-grained**.

There is no clear definition for the graph granularity, since it is possible to build nodes with just one unit generator, but this is not the common case. By a rule of thumb, fine-grained node graphs could be defined as node graphs with a significant node scheduling overhead.

2.2.1 Coarse-Grained Graph Parallelism

For coarse-grained graphs the scheduling overhead becomes less crucial, so each graph node can be scheduled as a task of its own. Letz et al [LOF05] proposed a dataflow scheduling algorithm which is implemented in Jack2. The idea is to distinguish between nodes that are 'ready', 'waiting' and 'executed'. Each node has an 'activation count', that equals the number of preceding nodes which have not been executed. At the beginning of each dsp cycle, the activation count is set to the number of direct predecessors, and nodes with an activation count of zero are scheduled for execution. After a node has been executed, it decrements the activation counts of all its succeeding nodes and if it drops to zero, they are scheduled as well.

2.2.2 Fine-Grained Graph Parallelism

Building signal graphs from unit generators, like creating SuperCollider synthdefs, results in very fine-grained graphs, since ugens can be very light-weight, maybe just requiring a few hundred to a few thousand CPU cycles (compare with Section 3.1). While a naïve approach would be to schedule each ugen on its own by traversing the graph in parallel, it would introduce a high scheduling overhead. On the other hand, statically scheduling ugens in a sequential order can be implemented quite efficiently [Puc91b], reducing the scheduling overhead to a minimum. Apart from the scheduling overhead, running related ugens serialized on one CPU may reduce cache misses, if the CPUs don't use a shared cache.

Ulrich Reiter and Andreas Partzsch [RP07] proposed an algorithm to cluster a fine-grained graph to a coarse-grained graph. The algorithm tries to find parallel sections in a DAG and to combine graph nodes to node clusters of similar CPU expenses. In order to achieve this, a "Processing Time Index" (PTI) is assigned to each graph node. For the best results, the maximum number of parallel sections matches the number of available CPUs. A practical problem with this algorithm is the requirement to know the CPU costs of each graph node. This may lead to a few substantial issues.

- The costs for a ugen may depend on the block size. Vectorizable nodes may be more efficient for block sizes of a multiple of the SIMD vector size.

- Especially for low-overhead unit generators, or wave-table oscillators, the cost may depend on memory locality. They could be considerably faster when the required memory regions are available in the CPU cache.

- Some unit generators do not have a constant algorithmic constant complexity. For example digital filters need to compute new filter coefficients when parameters like cutoff frequency or filter quality change. Since this requires calls to the math library, the costs may be significant. Other unit generators may have a parameter controlling the number of active voices, in these cases the required CPU costs would grow linearly with this parameter.

- Some unit generators for granular synthesis have a parameter to change the number of concurrent voices, changing the CPU costs linearly.

- The cost for a unit generator depends on the type of the CPU.

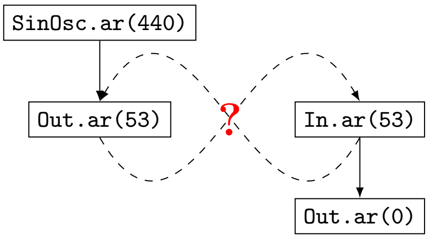

Figure 2.2: Example for an Ambiguous Dependency Graph

While the algorithm by Reiter and Partzsch works fine for simple signal flow graphs, it is hard to adapt the algorithm if the graph nodes access shared resources. In single-threaded applications, the resource access is ordered implicitly by the linearized (topologically sorted) graph. Apart from the signal flow, this implicit order would add another set of ordering constraints between all nodes which are able to access one resource. An example for this issue can be seen in the graph of Figure 2.2. A sine wave is generated and written to the bus number 53 in one part of the graph, a signal is read from this bus and then written to bus number 0. Without taking resource access into account, this graph could be split into two parts, one for the left and one for the right part, since there is no explicit dependency between them. When the graph is linearized, it would generate a structure in which either the left or the right side of the graph would be evaluated first, implicitly ordering to bus access either as 'write 53, read 53, write 0' or as 'read 53, write 0, write 53'. Both parts could be evaluated in parallel, if the `In` unit generator would read from another bus, like number 54.

While in this case, it could be statically analyzed that two parts access the same resources, it cannot be done in the general case, since the resources may be changed at run-time, depending on the unit generator, even at audio-rate. To cope with this, one would have to introduce an implicit order for all objects that are accessing a certain type of resource. This would add practical problems, since the algorithm requires some knowledge, of which nodes can access which type of resource. Depending on the use case, a huge number of implicit dependencies may be added to a signal graph, greatly reducing the parallel parts of the graph.

2.3 Thread Level Parallelism: Pipelining

As a fundamentally different approach, pipelining techniques can also be applied to sequential dependency graphs. When having a sequence of graph nodes which are supposed to be executed in a sequential order, nodes requiring the computational result of one of the earlier nodes can use the result of the signal block which has been computed during the last execution of the dependency graph [Dan08]. Figure 2.3 shows the behavior of a pipeline with 4 stages. During each clock tick, each pipeline stage processes one block of audio data and passes it to the next stage. Parallelism is achieved by computing each pipeline stage on a separate CPU. Since every pipeline stage delays the signal by one sample block, the pipelining technique is of limited use for low-latency applications. Since pipelining is usually defined statically, it doesn't necessarily scale well when increasing the number of CPUs. The execution time is bounded by the most expensive pipeline stage. So in order to achieve optimal speedup, all stages should require roughly the same

stage 1	block 1	block 2	block 3	block 4	block 5	block 6	block 7
stage 2		block 1	block 2	block 3	block 4	block 5	block 6
stage 3			block 1	block 2	block 3	block 4	block 5
stage 4				block 1	block 2	block 3	block 4

Figure 2.3: Pipeline with 4 stages

CPU time.

In order to reduce the latency of pipelining, each signal block can be split into smaller parts, which are sent through the pipeline. With this approach, the pipeline would have to be filled and emptied during each signal block. During the time that the pipeline needs to be filled or emptied, not all pipeline stages have work to do, which will limit the speedup. Smaller block sizes usually result in a higher scheduling and loop overhead, limiting the throughput even more. The lowest reasonable size for pipelined blocks would probably be the cache line size (16 samples on current CPUs). An experimental version of Jack2 implements pipelining with the constraints [OLF08] of splitting one Jack block to smaller blocks. Unfortunately, the developer team didn't publish any benchmarks of their approach.

In the early 1990s, static pipelining was implemented in FTS, running on the IRCAM Signal Processing Workstation. Recently, the same approach has been introduced into Pure Data. Both implementations share the same approach. The user can define subpatches, that should run on a different CPU. A more detailed discussion on these systems can be found in Section 6.1.

Chapter 3

Extending the SuperCollider Graph Model

Supernova introduces explicit node parallelism into the node graph model of SuperCollider. Explicit parallelism can also be found in systems the max-like systems Max/MSP (via `poly~`), Pure Data (via `pd~`) and it was implemented in FTS on the ISPW, although the approaches of the different systems are neither generic nor scalable (compare Section 6.1).

Csound has no notion of an explicit instrument graph. For a semantically correct parallelization, a dependency analysis is required to determine the parts of the instrument graph that can be executed in parallel [Wil09] (a more detailed discussion can be found in Section 6.2). For several reasons that are discussed in Section 3.2, this approach is not feasible for SuperCollider.

3.1 The SuperCollider Node Graph

Some overview on the SuperCollider node graph system was already given in Section 1.2.2. This Section will discuss the node graph and its implications in more detail. The examples assume two synthdefs, 'myGenerator', generating an audio signal, and 'myFx', that applies an effect on the signal. For the sake of simplicity, it is assumed that the busses used are hardcoded into the synthdefs, 'myGenerator' should write to a bus that is read by 'myFx', which itself writes to the bus number 0 (the first physical output).

The code in Listing 3.1 shows a simple use case with 4 generators and one effect synth, which is quite typical for using SuperCollider. Evaluating this code creates a group, inserts 4 synths of the type 'myGenerator' (`Synth.head` adds the synth to the head of the target group), and adds one effect synth ('myFx') after this group (via `Synth.after`). The corresponding node graph is displayed in Figure 3.1.

A group is defined as a linked list of nodes, which can be synths or groups. The SuperCollider

```
var generator_group, fx;
generator_group = Group.new;
4.do {
    Synth.head(generator_group, \myGenerator)
};
fx = Synth.after(generator_group, \myFx);
```

Listing 3.1: Using SuperCollider's Group class

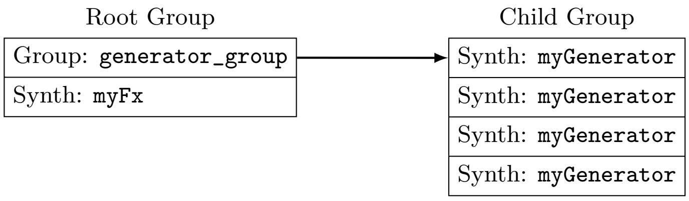

Figure 3.1: Node Graph for Listing 3.1

server provides a fixed **root group**, that serves as root for the node hierarchy, and which is used as default target if the user does not provide a target when creating a node (like when creating the group via `Group.new` in the example). So the node hierarchy actually models a tree structure with synths (or empty groups) as its leaves. Groups have some semantic implications, which are all related to a structured audio synthesis:

Structuring the audio synthesis

Groups function as a nested container for other nodes. They can be used to build a structure for an audio synthesis application, and they provide a high-level interface to free all child nodes or recursively free all synths while keeping the group structure untouched.

Order of execution

Groups define the order of execution of their child nodes. When a group is executed, the nodes are evaluated from head to tail, recursively traversing child groups. In this example, first the group `generator_group` would be evaluated, by evaluating each `myGenerator` instance from head to tail, and then the `myFX` synth would be evaluated. In this example, we need to make sure that the `generator_group` is actually evaluated before the effect synth.

Node addressing

Groups can be used to address all children with a single command. This is especially handy when setting control parameters for multiple nodes.

When a new node is added to the node hierarchy, its position is specified by a reference node and a relation. The relations are `addToHead` and `addToTail`, adding the new node to the head or the tail of a reference group, `before` and `after`, adding a node before or after a reference node, or `replace`, replacing a reference node.

3.2 Automatic Parallelism for SuperCollider?

SuperCollider's groups are sequential by definition. There is no automatic dependency analysis (like with instruments in Csound or with tilde objects in max-like systems), but the user is responsible for defining the correct order of execution. In SuperCollider, this wouldn't even be possible. The reason for this is the way that resources are accessed by unit generators. While in Csound instruments communicate by using global variables, or a built-in bus system, SuperCollider uses only busses for synth communication. Buses are addressed by integer values, that are encoded into signals and passed to the arguments of ugens. Unit generators are loaded as plugins into the SuperCollider server and this is no exception for ugens that are accessing busses (or buffers). The server has no means to determine whether a ugen is accessing a specific bus or buffer.

```
var generator_group, fx;
generator_group = ParGroup.new;
4.do {
    Synth.head(generator_group, \myGenerator)
};
fx = Synth.after(generator_group, \myFx);
```

Listing 3.2: Using the Supernova ParGroup class

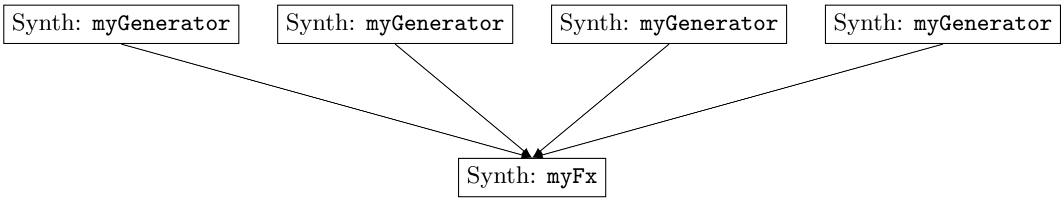

Figure 3.2: Dependency Graph for Listing 3.2

Theoretically it would be possible to add some information about the resources that a synth is accessing to the internal data structures. However there would be two cases that could not be handled by this approach. First, the resource index could be controlled by another synth, so it would not be possible to determine the resources that are accessed by a synth before the synth is evaluated. It would even be possible, that the signals encoding the resource indices run at audio rate, making an automatic dependency analysis at control-rate impossible. The only workaround (that can actually be found in the parallel version of Csound, compare Section 6.2), would be to execute all synths sequentially, unless the resource access pattern can be inferred before the control-rate tick. For practical applications, this would greatly reduce the parallelism.

3.3 Parallel Groups

However, the SuperCollider node graph can be extended to specify parallelism explicitly. This approach has been implemented in Supernova, by introducing the concept of **parallel groups**. A parallel group behaves similar to a group, with the exception, that group members are not ordered and therefore can be evaluated in parallel. At the scope of the language, parallel groups are represented by the `ParGroup` class, that provides the same interface as the `Group` class.

The example in Section 3.1 can easily be adapted to use a parallel group instead of a sequential group, since we assume that `myGenerator` instances do not depend on each other. Listing 3.2 shows how this would be implemented. In this case, it is as simple as replacing the `Group` instance with a `ParGroup`. The node graph looks fundamentally different though, modeling a directed acyclic dependency graph, as shown in Figure 3.2.

The strength of this approach is its simplicity. Parallel groups extend the SuperCollider node graph with a simple concept and integrate well into existing code. It also guarantees backwards and forwards compatibility with scsynth. Since the `ParGroup` class has the same interface as the `Group` and its only semantic difference is the missing ordering constraint for its child nodes, it is safe to replace the `ParGroup` class with the `Group` class to run code on scsynth. On the scsynth level, it may even be possible to introduce an OSC handler for the command that is used for the creation of parallel groups, as an alias for the creation of groups. Backwards compatibility is trivial, since Supernova does not change any of the semantics of scsynth, but just provides an

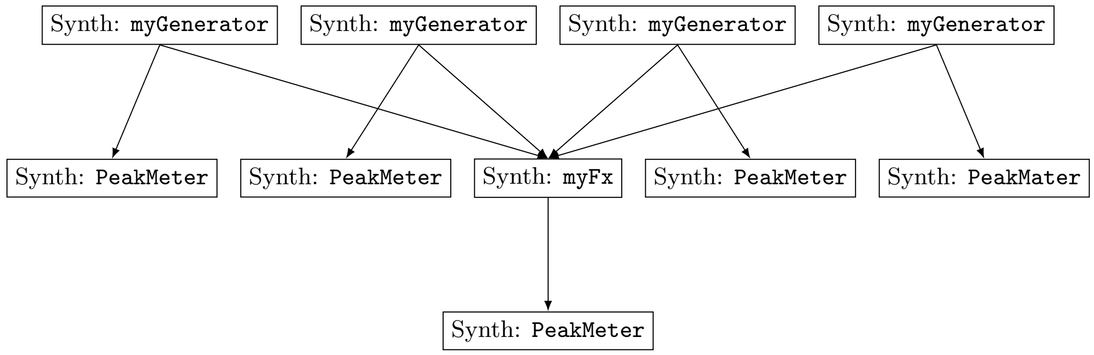

Figure 3.3: Semantically Correct Dependency Graph

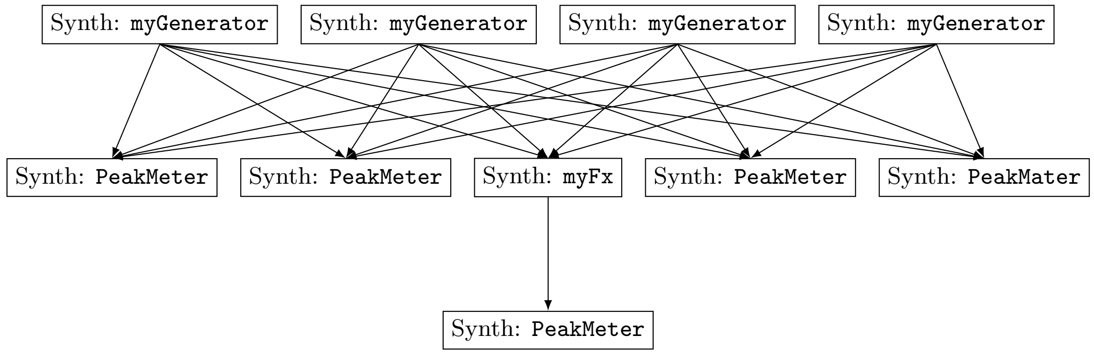

Figure 3.4: Dependency Graph with Parallel Groups

extension. The concept of parallel groups also fits well into the idea of an explicit handling of the node graph.

When adding a node to a parallel group, the relations `addToHead`, `addToTail`, `before` and `after` only have limited meaning, since the elements of a group are not ordered. Instead, a node would simply be inserted into the parallel group. To increase the expressive power, a new relation `insert` has been introduced. While it is sematically equivalent with `addToHead`, it should increase the readability of the code.

3.3.1 Limitations

However there are some limitations of this approach, since it doesn't provide a true dependency graph. As an example, the dependency graph in Figure 3.3 will be discussed, which may arise from a use case where peak meters are run after each synth to update the VU meters of a GUI.

To formulate this use case with parallel groups, the peak meters for the generator synths could be grouped with the effect synth as shown in Figure 3.4, introducing a number of rather unnecessary node dependencies. The peak meter for a single synth doesn't only depend on this synth, but on all the other synths in the parallel group. In cases where the different synths require a different amount of CPU time, the first peak meter is not allowed to start before the last generating synth finishes.

```
var fx = Synth.new(\myFx);
4.do {
    var synth = Synth.preceding(fx, \myGenerator)
    Synth.succeeding(synth, \PeakMeter)
};
```

Listing 3.3: Using the proposed satellite nodes

3.4 Proposed Extension: Satellite Nodes

The limitation that has been described above, could be resolved by introducing another extension to the node graph concept, which we call **satellite nodes**. Unlike nodes in groups or parallel groups, which have depedency relations to both predecessors and successors, satellite nodes have just one dependency relation to a reference node. Satellite successors only depend on their reference nodes, while each reference node depends on the execution of their satellite predecessors. Satellite nodes can be nested, so that they can have satellite nodes themselves, but nevertheless, they will be in the dependency relation with only one node in the tree hierarchy.

With this extension, the example of the previous section, as it is shown in Figure 3.3, could be implemented with correct dependencies, since the `PeakMeter` synths could be defined as successors of the corresponding synths (compare Listing 3.3). Since also the generator synths can be specified as satellite nodes of the effect synth, this use case can be specified without the need to use parallel groups.

Unlike parallel groups, satellite nodes do not fit into the node graph concept in an obvious manner. Since no equivalent concept exists in SuperCollider, emulating them with the available means of scsynth is rather difficult, mainly because one need to ensure that satellite nodes would be moved in relation with their reference node. The most important semantic aspects that need to be defined are addressing and lifetime. In the tree hierarchy, satellite nodes are located on the same level as their reference nodes, so that they can be addressed by the parent group of their reference when setting control parameters. The lifetime of a satellite node is governed by their reference nodes. If a node is freed, all their satellites are freed as well. Likewise, satellite nodes are paused if their reference nodes are paused.

3.5 Alternative Approach: Freeform Node Graphs

A fundamentally different approach would be to formulate the node graph with explicit dependencies. The approach of using parallel groups is simple, since it easily integrates into the concept of the SuperCollider node graph and even keeps compatibility. Free node predecessors and successors would extend the node graph in a incompatible way, but still keep a node hierarchy, although it would enable the user to formulate a more fine-grained hierarchy.

It would be possible to completely get rid of the node hierarchy, breaking backwards and forwards compatibility with scsynth. In the current approach, all nodes have a parent node, except for the root group that is the root of the node hierarchy. It would be possible to break this limitation, though. A node graph could be formulated in a way, that all dependencies between nodes would have to be specified explicitly, and that multiple dependencies may be possible. In all current approaches, it is not possible to specify a dependency that node `A` should be evaluated after node `B`, but before node `C`, except by using a (sequential) group. If these node were ‘free nodes’, it would be possible to specify their dependency explicitly. While this at first seems to be appealing, it would introduce quite a number of problems. If dependencies can be specified explicitly, it would easily be possible to introduce cycles into the dependency graph, which would not be allowed, since the dependency graph should by definition by acyclic. At the scope of the language, an additional functionality would have to be developed, to specify both node positions (in groups) and node dependencies. This would significantly increase the complexity of formulating node graphs.

While a freeform node graph would be of theoretical interest, since it would reduce the number of node dependencies that limit parallelism to a minimum, the complexity of its formulation and lack of node hierarchy would make it difficult to introduce it into the SuperCollider system.

Chapter 4

The Implementation of Supernova

> Taking any kind of lock even a spinlock is a no-no on a real time thread such as the CoreAudio I/O thread. You then have no guarantee that you will meet your deadline, and if you take too long the Mac OS X scheduler will degrade your priority
>
> *James McCartney* [McC09]

Supernova is designed as replacement for the SuperCollider server scsynth. This Chapter covers the important aspects of its implementation. A discussion of the real-time aspects of the implementation is done in Section 4.1. The next Section describes the node graph and DSP queue subsystem, which is the main contribution of this master thesis. Section 4.3 explains the API extensions for the SuperCollider unit generator interface. The last Section, 4.4, describes the concepts of a generic SIMD library, that is not directly part of Supernova, but has been developed as a side project and is heavily used by SuperCollider's unit generators.

4.1 Parallelism in Low-Latency Soft Real-Time Systems

The usual goal when parallelizing an application it to increase its throughput. However, commonly used techniques like pipelining introduce a trade-off between throughput and latency. Computer Music Systems have special requirements, since they are working in soft real-time contexts with demands for low worst-case latencies. Therefore they have specific demands for hardware, operating system and software design.

4.1.1 Real-Time Audio Threads, Wakeup Latency

The architecture of Supernova heavily relies on a low wakeup latency of high-priority threads. The **wakeup latency** is defined as time interval between signaling and the execution of a thread. In the beginning of each audio callback, the helper threads need to be signaled in order to work on the job queue. For a useful application, the wakeup latency needs to be significantly lower than the duration of one block (1.3 ms for 64 samples at 48000Hz).

Real-Time Preemption Kernel Patches

For several years, there has been a set of real-time preemption patches for the linux kernel, maintained mainly by Ingo Molnar and Thomas Gleixner, implementing features which reduce the latency of the kernel. The patchset is slowly being merged into the mainline kernel. The real-time preemption patches rely mainly on the following techniques to reduce the latency:

Sleeping Spinlocks

Spinlocks are commonly used to synchronize data structures in the linux kernel. Many of these spinlocks are converted to mutexes. Acquiring a lock may preempt the calling thread, and also the guarded critical sections may be preempted. An additional API for 'real' spinlocks (`raw_spinlock`) has been introduced in order to provide non-preemptible critical sections.

Priority Inheritance

Priority inversion happens, if a low-priority thread holds a shared resource, that is required by a high-priority thread. In this case, the high-priority thread is blocked until the low-priority thread releases the resource, reducing its priority effectively to the priority of the low-priority thread. To avoid this behavior, the RT Preemtion patches implement Priority Inheritance. The priority of the low-priority thread is raised to the value of the high-priority thread, until it releases the shared resource.

Threaded Interrupts

The soft interrupt handlers are converted to preemptible kernel threads. It is possible to assign priorities to these interrupt handler threads which may be lower than the priorities of user-space threads which are running with a real-time scheduling policy.

Wakeup Latency of Real-Time Threads

The latency of real-time threads is mainly dominated by hardware effects. The most important ones are [Boh09]:

System Management Interrupts (SMI)

System Management Interrupts are generated by the hardware. They have the highest interrupt priority of the system and may consume several hundred microseconds, and are not visible to the operating system. The SMI code is stored in the BIOS and used for system-specific operations such as power management (like CPU frequency scaling or fan control), hardware emulation (e.g. emulation of PS/2 mice or keyboards from USB devices) or error handling (memory/chipset errors, thermal protection).

DMA Bus Mastering

DMA devices can cause bus mastering events, which may introduce CPU stalls of many microseconds. Like SMIs, this is a hardware-specific issue, unrelated to the operating system.

CPU scaling

Modern CPUs can change their clock frequency and core voltage on the fly. When using a scaling governor that dynamically changes the CPU speed, additional latencies are introduced.

Beside hardware effects, there are some issues related to the operating system that introduce additional latencies. For a long time, the biggest issue with the Linux kernel is the Big Kernel Lock (BKL), a global spinlock, that is used by different subsystems and can therefore introduce large latencies. The BKL was slowly removed from the linux kernel, but it hasn't been completely removed before version 2.6.39, which is expected to be released in summer 2011.

The wakeup latency can be measured with the program 'svsematest', testing the latency of SYSV semaphores, which is available from the RT-tests repository. My personal reference machine, an Intel Core i7 920, can achieve worst-case scheduling latencies of 12 µs with kernel

2.6.33-rt4, if power management, frequency scaling and simultaneous multithreading are disabled, with an average of 2 µs. Enabling power management and CPU scaling, the worst-case scheduling latency raises to about 200 µs. Enabling SMT, it raises to about 250 µs. My second machine, a Thinkpad T60 laptop with an Intel Core2, can be tuned to just below 100 µs. For MacOSX, Letz et al [LOF05] suggests a worst-case scheduling latency of between 200 and 250 µs.

4.1.2 Memory Management

For user-space applications, dynamic memory management is an important issue, since a real-time thread may be preempted, because `malloc` is usually not implemented in a lock-free manner and may issue system calls to allocate memory from the operating system. There are several publications about concurrent memory allocators [BMBW00, SAN06, DG02, GM], even two about lock-free allocators [Mic04, GPT05], but these still have the need to directly allocate memory from the operating system, so they are not necessarily suited for real-time applications.

Remy Boehmer [Boh09] suggests that dynamic memory management of real-time applications could be achieved by tuning the GLIBC implementation of `malloc`, so that it neither allocates nor frees memory from/to the operating system. Since the malloc implementation of GLIBC is not lock-free, this approach is not suitable for multi-threaded real-time applications, though.

There is one publication about TLSF [MRCR04], a real-time safe memory allocator, which has a bounded response time of O(1), using a pre-allocated memory pool. Since it uses a pre-allocated memory pool, it doesn't need to dynamically allocate memory from the operating system, but this comes at the cost that one needs to allocate all required memory in advance. The main drawback of TLSF is, that it is designed as a single-threaded memory allocator.

Supernova makes heavy use of memory pools, managed by TLSF. Since the DSP helper threads don't need to allocate memory, it is safe to use one memory pool for real-time memory, that is accessed only by the main DSP thread. In certain use cases, data needs to be transferred from the real-time thread to a non real-time thread, though. In these cases, the memory is not directly freed to the memory pool, but pushed to a lock-free free-list that is accessed from the real-time thread. Using this indirection, the memory pool is only accessed from the real-time thread. This comes at the cost of at least two additional compare-and-exchange instructions per freed object (at least one in the non real-time and one in the real-time thread).

Beside memory allocation, page faults are a second source of memory-related latencies [Boh09]. There are **major page faults**, which require the operating system to load a swapped page from a hard disk, causing long latencies, and **minor page faults**, that are caused, when the requested page can be mapped without IO access. This is mainly the case with freshly allocated pages, but could also happen if the process tries to write to a copy-on-write page. Major page faults can be avoided by locking the pages of a process to physical RAM using operating-system specific system calls. On POSIX-compliant systems, the `mlock` and `mlockall` functions may be used to achieve this. Minor page faults cannot be avoided, but it is possible to avoid them during real-time critical sections of a program by writing some data to the pages before they are accessed from the real-time context.

4.1.3 Synchronization Primitives

In general, real-time threads should not use any blocking synchronization primitives. For years audio applications have been designed in a way. that they don't wait for locks in the audio threads. Systems like Pure Data that use a global system lock have been considered as not real-time safe. Concurrent real-time applications can use blocking locks under certain constraints, though. The real-time preemption patches implement priority inheritance to reduce latencies, introduced by priority inversion.

For computer music applications, the wakeup latency is already quite significant, so the interaction with the process scheduler of the operating system should be avoided. For synchronization between real-time threads, spin-locks can be used under certain constraints:

- No synchronized thread is preempted
- Locks are acquired very rarely
- The guarded sections are small (busy waiting is not very expensive)

The first constraint would imply that the synchronized threads have a similar (possibly real-time) priority and that the number of synchronized threads does not exceed the number of CPU cores. It could also be achieved by pinning the threads to different physical CPUs. The second condition can be relaxed, if reader-writer spinlocks are used and write access is acquired very rarely.

If possible, lock-free data structures should be used to avoid locks completely. During the development of Supernova we have implemented several lock-free data structures. These data structures have been submitted for inclusion in the Boost C++ libraries and will undergo a peer review in Summer 2011[1]. In Supernova, several lock-free stacks and queues are used for passing objects between real-time and non real-time threads and for synchronizing real-time threads.

4.1.4 Real-Time Safe C++

C++ is a decent choice for implementing real-time applications, if certain care is taken, when using the language. In the standardization process of the new revision C++0X, a technical report on the performance of C++ [Gol06] has been published, giving a detailed timing analysis of different language features. The main language-specific problems for real-time systems are related to run-time type information (RTTI) and exception handling. When using the RTTI for a down-cast or cross-cast via `dynamic_cast`, the execution time depends on the position of source and destination classes in the class hierarchy, so it is not possible to define a general rule for the worst-case execution time. Exception handling itself introduces a small run-time overhead. The execution time analysis of exception handling depends on the implementation of exceptions, which may vary among different compilers. Since computer music systems are not hard but soft real-time systems exceptions can be treated as cases, where the deadline may be missed.

The memory management of the Standard Template Library (STL) may cause bigger problems when implementing real-time systems in C++. While the impact of dynamic memory management on real-time systems in general have already been discussed in Section 4.1.2, the semantics of C++ allocators make it rather difficult to solve the issue by using memory pools. C++ allocators define an interface, how containers allocate and free memory. All STL containers can be configured at compile-time to use a specific class as its allocator. Instances of allocator classes are defined to be state-less: this means that a memory chunk that is allocated from one instance may be freed to another instance. This implies that one needs to implement a new class for each memory pool that one wants to use as a specific memory pool. Implementing thread-local memory pools with allocators is therefore rather difficult. Fortunately, there is boost.container, an alternative implementation of STL-style containers, that support stateful allocators, which is queued for review to be included in the boost library.

[1] The review of this proposed `boost.lockfree` library is partially funded by Google Inc. as part of the Google Summer of Code 2011.

Supernova tries to overcome this limitation by providing one memory pool that is only used by containers of the main audio thread. STL containers are not passed between non real-time and real-time threads, and the real-time helper threads do not allocate any memory at all.

4.2 Dsp Thread Subsystem

The dsp thread subsystem is the heart of Supernova and the part that is fundamentally different from the traditional SuperCollider server scsynth. The architecture itself is rather simple, the tricky part is the real-time safe implementation. Unlike scsynth, Supernova distinguishes between a representation of the **node graph** and the **dsp queue**.

4.2.1 Node Graph

The server contains a representation of the SuperCollider node graph, as described in Section 3.1. The node graph usually has no client-side representation, but the client uses OSC commands to query and modify the server. It is implemented as recursive data structure with a (sequential) group as root node and synths, groups and parallel groups as children. The `abstract_group` class, from which both the `group` and the `parallel_group` class are derived, manages its children by using two linked lists, a list of its child nodes and a list of its child groups. For sequential groups, both lists form a skip-list, improving the performance in some parts of the algorithm for generating the dsp queue.

4.2.2 Dsp Queue

In scsynth, the node graph can be used directly for the audio synthesis, since it can efficiently be iterated. This does not apply for the node graph in Supernova, because of its notion of parallel groups. Instead, the node graph is converted to a **dsp queue**, basically a compact representation of the node graph, that is suited for driving Supernova's **dsp thread interpreter**. In the queue representation, the notion of synths and groups does not exist any more, but is replaced by **dsp queue items**. The dsp queue is treated as a coarse-grained graph (compare Section 2.2.1), which is scheduled with an algorithm similar to the one by Letz et al [LOF05]. Each queue item maintains an atomic **activation count** and a list of successor queue items. The queue class itself contains a list of references to all queue items and a list of initially runnable items. After a queue item is executed, the activation count of each successor is decremented, and it if drops to zero, the item is scheduled for execution (compare Algorithm 4.1).

Usually this is done by adding the queue item to a lock-free stack. In order to reduce the contention of this stack, the first element is directly selected to be the next item to be evaluated by the calling thread. This avoids at least two compare-and-exchange instructions.

Algorithm 4.1 Run Dsp Queue Item

```
Evaluate Queue Item
for all $i$ in successors do
  $i.activation\_count \leftarrow i.activation\_count - 1$
  if $i.activation\_count = 0$ then
    add $i$ to scheduler queue
  end if
end for
```

Dsp Queue Generation

The dsp queue is generated by traversing the node hierarchy in reverse, starting at the root group. Child groups are traversed recursively. The queue generation for parallel groups and for sequential groups differs because of their different semantics. For parallel groups, a dsp queue node is created for each direct child synth. If parallel groups contain other groups, they are traversed recursively, ensuring the correct list of successor jobs. For sequential groups, the queue nodes are optimized in a few ways to combine various synths into a single queue node to avoid the scheduling overhead for sequential synths.

Since the dsp queue generation is done synchronously to the audio thread, it is required to be implemented in a very efficient manner to avoid audio dropouts. In order to achieve this, several optimizations have been introduced. First of all, the number of dynamic memory allocations was reduced to a minimum. The algorithm needs to allocate some memory from the real-time memory pool, but the number of allocations has been reduced by allocating a memory chunk that is big enough to hold the memory for all queue nodes.

4.2.3 Dsp Threads

The dsp queue is interpreted by several dsp threads: the main audio thread, which is managed by the audio driver backend, and a number of helper threads, that are woken when the backend triggers the main audio thread. The total number of audio threads is limited by the number of physical CPUs and threads are pinned to different CPUs.

Algorithm 4.2 shows the behavior of the main dsp thread. When called, it dispatches callbacks, e.g. to handle OSC commands, wakes the dsp helper threads and adds all initially runnable queue items to the lock-free stack. The dsp threads then poll the stack of runnable items until all items have been executed (compare Algorithm 4.3).

This approach may cause busy waiting, if there are fewer items to be processed in parallel than available processors. This behavior has been implemented because of the overhead in wakeup latency. If the wakeup latency would drop reliably below 10 to 20 microseconds, semaphores could be used to put real-time threads to sleep, to avoid busy waiting.

Algorithm 4.2 Main Audio Thread

```
dispatch callbacks (handle OSC commands, etc)
dispatch scheduled OSC messages
wake helper threads
add initially runnable jobs to scheduler queue
while some queue items are remaining do
  poll scheduler queue
  if got queue item then
    run queue item
  end if
end while
```

4.3 Extending the SuperCollider Plugin Interface

SuperCollider bundles a huge number of unit generators for all kinds of applications: simple building blocks for audio synthesis like oscillators, filters, sampling ugens, more complex ones like reverbs or frequency domain processors and even some highly specialized machine-listening

Algorithm 4.3 Audio Helper Threads

```
while some queue items are remaining do
  poll scheduler queue
  if got queue item then
    run queue item
  end if
end while
```

```
void In_next_a(IOUnit *unit, int inNumSamples)
{
    [...]

    for (int i=0; i<numChannels; ++i, in += bufLength) {
        int32 busChannel = (int32)fbusChannel + i;
        ACQUIRE_BUS_AUDIO_SHARED(busChannel);
        if (touched[i] == bufCounter)
            Copy(inNumSamples, OUT(i), in);
        else
            Fill(inNumSamples, OUT(i), 0.f);
        RELEASE_BUS_AUDIO_SHARED(busChannel);
    }
}
```

Listing 4.1: Using the unit generator API extension

ugens. There is also the sc3-plugins project[2], a repository of 3rd-party unit generators that are contributed by various people. Some of these may eventually be included in the main SuperCollider distribution.

The unit generators are dynamically loaded plugins, which are loaded via a basic C-style interface when the synthesis server is launched. When a plugin is loaded, the contained unit generators are registered in the server application by storing function pointers to ugen constructor and destructor functions. The actual ugen processing function is then selected in this constructor function at the time of synth creation. The server passes a struct of function pointers to the plugin, which is used to call functions of the server from within the plugin code. This design decision helps a lot when trying to load SuperCollider ugens into another application, by introducing a run-time overhead of a pointer indirection for each function call from the plugin code into the host.

Beside this interface table, there is a global singleton `World` struct, containing all the run-time information that is accessible from the unit generator, like sampling rate, the buffers used for storing buffers and busses, random number generator and the like. A pointer to this struct is stored in every unit generator instance.

The only difference in the ugen API between scsynth and Supernova is the requirement for resource synchronization. The resources that need to be guarded are busses and buffers, which is done by some custom reader-writer spinlocks. To implement this, two structs need to be changed. The `World` struct and the `SndBuf` struct, containing the information of sample buffers, need to be extended to include a reader-writer spinlocks for each audio-rate busses and sample buffer. control-rate busses don't need to be synchronized, since they are written atomically and

[2] the project's website is available at http://sc3-plugins.sourceforge.net/

`ACQUIRE_BUS_AUDIO(INDEX)`	Acquire writer-lock for audio bus
`RELEASE_BUS_AUDIO(INDEX)`	Release writer-lock for audio bus
`ACQUIRE_BUS_AUDIO_SHARED(INDEX)`	Acquire reader-lock for audio bus
`RELEASE_BUS_AUDIO_SHARED(INDEX)`	Release reader-lock for audio bus
`ACQUIRE_SNDBUF(INDEX)`	Acquire writer-lock for buffer
`RELEASE_SNDBUF(INDEX)`	Release writer-lock for buffer
`ACQUIRE_SNDBUF_SHARED(INDEX)`	Acquire reader-lock for buffer
`RELEASE_SNDBUF_SHARED(INDEX)`	Release reader-lock for buffer
`LOCK_SNDBUF(INDEX)`	Create RAII-style writer lock object for buffer
`LOCK_SNDBUF_SHARED(INDEX)`	Create RAII-style reader lock object for buffer
`LOCK_SNDBUF2(INDEX1, INDEX2)`	Create RAII-style writer lock object for 2 buffers
`LOCK_SNDBUF2_SHARED(INDEX1, INDEX2)`	Create RAII-style reader lock object for 2 buffers
`LOCK_SNDBUF2(INDEX1, INDEX2)`	Create RAII-style writer lock object for 2 buffers
`LOCK_SNDBUF2_SHARED_EXCLUSIVE(INDEX1, INDEX2)`	Create RAII-style lock object for 2 buffers, creating a reader-lock for the first and a writer lock for the second buffer. Deadlock safe.
`LOCK_SNDBUF2_EXCLUSIVE_SHARED(INDEX1, INDEX2)`	Create RAII-style lock object for 2 buffers, creating a writer-lock for the first and a reader lock for the second buffer. Deadlock safe.

Table 4.1: Unit Generator API Extension Macros

therefore cannot contain inconsistent data.

Because of the limited API changes, unit generators that are compiled for SuperCollider are binary compatible with Supernova, unless they access audio busses or signal buffers. The number of unit generators that access audio-rate busses is quite limited. But there are various ugens that access signal buffers like sampling or delay ugens or wavetable oscillators. Also the phase vocoder ugens use signal buffers for storing spectral frames.

In order to avoid deadlocks while resource locking, two locking policies need to be followed. First, a unit generator is only allowed to lock one type of resource. This rule is rather trivial, since there does not seem to be a single unit generator that is accessing both busses and buffers. The second policy introduces a locking hierarchy. If a unit generator needs to lock multiple resources simultaneously, it should acquire the resources in ascending order of their indices, in order to avoid circular wait conditions.

Depending on the use case, the contention may be reduced by formulating the graph accordingly. Instead of playing with several different synths to one summing bus, each synth could play to a separate bus, which could be summed by another synth.

The resource locking code is implemented with several C preprocessor macros. The use of these macros makes it easy to conditionally enable the locking code in order to compile the unit generators for Supernova or to disable them for scsynth. Listing 4.1 shows the modified versions of the `In.ar` unit generator. A reader-lock of an audio bus is acquired and released using the macros `ACQUIRE_BUS_AUDIO_SHARED` and `RELEASE_BUS_AUDIO_SHARED`. Table 4.1 shows a full list of all API extension macros.

The API extension and the modified unit generators have already been merged into the SuperCollider source.

4.4 Nova SIMD

During the development of Supernova, a generic SIMD framework was developed. It is based on code written for my former project Nova [Ble08], but has been reimplemented from scratch in a generic way using C++ templates. This library has been integrated into SuperCollider, speeding up several vectorizable operations.

The major problem when trying to use SIMD instructions is portability. Neither C nor C++ expose a SIMD interface as part of the language. The GCC compiler provides a vector extension for C, but its interface is rather limited, supporting just simple arithmetic and bitwise operators. In particular, it is lacking support for element-wise compare operators, which are required for branch elimination. In order to make use of SIMD instructions, one would have to use libraries like liboil, compiler-specific intrinsics or assembler code to make use of the SIMD hardware. These days, the main SIMD instruction set is the family of **Streaming SIMD Extensions (SSE)** of Intel architectures IA32 and x86-64. **Altivec**, an extension to the PowerPC instruction set, has been widely used in Apple's personal computers, but since Apple's transition to Intel CPUs, Altivec instructions have lost their importance for consumer applications. Future Intel CPUs will introduce a new instruction set extension called **Advanced Vector Extension (AVX)**, doubling the width of SIMD registers from 128 to 256 bit. Since the SSE instruction set has been continuously extended, there are some practical problems when distributing programs in binary form. For full compatibility with old CPUs, SSE instructions should not be used at all. For x86-64, one can assume that all CPUs support the SSE and SSE2 instruction sets. The only possibility to make use of all instruction set extensions by keeping compatibility with old CPUs would be to introduce a run-time dispatching. However, depending on the application, it may be difficult to implement. Some recent ARM CPUs support the **NEON** instruction set, providing SIMD instructions for integer and single-precision floating point types.

The only aspect that all SIMD instructions have in common, is that they process multiple data in one instruction. The number of data per instruction depends on the instruction set and the data type. The SSE instruction set family for example provides 128-bit registers, which can contain 2 double-precision floating point numbers, 4 single-precision floating point numbers or 32-bit integers, 8 16-bit integers or 16 8-bit integers.

4.4.1 The vec Class

The goal of the Nova SIMD framework is a generic API that can be used as building block for the implementation of vectorized functions. It is built upon one template `vec` class, which represents one SIMD vector. Using template specialization, the class can be implemented for specific instruction sets. The basic synopsis can be found in Listing 4.2. Besides basic functions for initialization, memory loads and stores and element-wise access, it provides two static members, that help when implementing generic algorithms with this class. The `size` member specifies the number of elements per SIMD vector and `objects_per_cacheline` gives a hint, how many elements would fit typically fit into a single cache line. These two static members make it easy to formulate a high-level SIMD algorithm by still making use of hardware-specific information. The algorithm itself can make use of features which may be provided by some instruction sets, but would have to be emulated on others.

The `vec` class is currently implemented with the SSE and SSE2 instruction sets for single- and double-precision floating point numbers. Support for NEON is currently under development.

```
template <typename float_type>
class vec
{
public:
    static const int size;
    static const int objects_per_cacheline;

    vec(void);
    explicit vec(float_type f);
    vec(vec const & rhs);

    void load(const float_type * data);
    void load_first(const float_type * data);
    void load_aligned(const float_type * data);

    void store(float_type * dest) const;
    void store_aligned(float_type * dest) const;
    void store_aligned_stream(float_type * dest) const;

    void clear(void);
    void set(std::size_t index, float_type value);
    void set_vec(float_type value);

    float_type get (std::size_t index);
};
```

Listing 4.2: Basic interface of Nova SIMD's **vec** class

4.4.2 Nova SIMD algorithms

On top of this vec class, several algorithms have been implemented, starting from simple arithmetic vector functions, functions for mathematical operations, but also algorithms that are specific for audio signal processing, like for peak metering or unit conversions.

The mathematical functions are based on the algorithms that are found in the Cephes math library [Mos92]. In order to vectorize the algorithms, all branches need to be eliminated. This is usually done by computing the results for all branches and using bitmasks to select the desired result. In a similar way, range reduction for the arguments is performed. For some cases, the coefficients for the approximation polynomials have been refined with Sollya [CJL19]. Listing 4.3 shows an example how the vec class can be used for implementing a vectorized `tanh` function for single-precision floating point numbers. While all approximation polynomials are currently computed with Horner's scheme, it may be interesting to provide an alternative implementation using Estrin's scheme, which should make better use of the instruction-level parallelism on out-of-order CPUs.

```
template <typename VecType>
inline VecType vec_tanh_float(VecType const & arg)
{
    const VecType sign_arg = arg & VecType::gen_sign_mask();
    const VecType abs_arg  = arg ^ sign_arg;
    const VecType one      = VecType::gen_one();
    const VecType two (2);
    const VecType maxlogf_2 (22);
    const VecType limit_small (0.625f);

    /* large values */
    const VecType abs_big          = mask_gt(abs_arg, maxlogf_2);
    const VecType result_limit_abs = one;

    /* small values */
    const VecType f1(-5.70498872745e-3);
    const VecType f2( 2.06390887954e-2);
    const VecType f3(-5.37397155531e-2);
    const VecType f4( 1.33314422036e-1);
    const VecType f5(-3.33332819422e-1);

    const VecType arg_sqr = abs_arg * abs_arg;
    const VecType result_small = ((((f1 * arg_sqr
                                     + f2) * arg_sqr
                                    + f3) * arg_sqr
                                   + f4) * arg_sqr
                                  + f5) * arg_sqr * arg
        + arg;

    const VecType abs_small = mask_lt(abs_arg, limit_small);

    /* medium values */
    const VecType result_medium_abs = one -
        two / (exp(abs_arg + abs_arg) + one);

    /* select from large and medium branches and set sign */
    const VecType result_lm_abs = select(result_medium_abs,
                                         result_limit_abs,
                                         abs_big);
    const VecType result_lm = result_lm_abs | sign_arg;

    const VecType result = select(result_lm, result_small, abs_small);

    return result;
}
```

Listing 4.3: Using the `vec` class to implement `tanhf`

Chapter 5

Experimental Results

This chapter discusses the performance results that can be achieved by using Supernova instead of scsynth. When evaluating the performance of Supernova, the interesting aspect is the worst-case execution time of signal processing engine. The signal processing engine is triggered once for each signal vector. For low latency applications, the execution time for this operation should never exceed the time that this signal vector represents (for 64 samples at 48000 kHz, this would be 1.3 ms), otherwise the deadline for delivering audio data to the backend would be missed. Some audio backends compute multiple control rate blocks during one backend tick. While increasing the latency of the audio signal, this reduces some overhead in the backend. For Supernova, this relaxes the deadline for control rate ticks. Because all control rate ticks of a backend tick need to be evaluated before the deadline of the backend tick, some deadline misses for control rate ticks can be tolerated.

5.1 Experimental Setup

The tests were carried out on an Intel Core i7 920 quad-core machine, running Linux, that was tuned for real-time performance (compare with section 4.1.1). The benchmarks themselves have been done by measuring time differences between `clock_gettime` system calls. For running the benchmarks, the clocksource was configured to use Time Stamp Counters (TSC) [Intb]. Since CPU frequency scaling has been disabled, they provide accurate values.

The measured durations are sampled and stored in a histogram with nanosecond accuracy. Compared to simple throughput benchmarks, the use of histograms has certain advantages. In particular, the histograms show not only the average-case, but also the worst-case behavior, so they can be used to identify latency hotspots. Some other interesting information can be obtained by examining the distribution of the histograms.

5.2 Engine Benchmarks

The first part of the benchmarks focuses on the scalability of the engine. It is impossible to provide reasonable benchmarks for each possible use case of Supernova. The benchmarks therefore concentrate on different use cases that stress different parts of the implementation.

For running the benchmarks, two synths with different CPU costs are used (see Listing 5.1). They are based on the `FSinOsc` unit generator, which implements a sine wave oscillator via a ringing filter. Because its CPU costs do not depend on its argument or its internal state, it is a suitable unit generator for benchmarking the engine. The lightweight synth instantiates 4

```
SynthDef(\small, {|out|
    var sig = FSinOsc.ar(440!4); // generate 4 Fast Sine Oscillators
    Out.ar(out, sig)             // write a one-channel mixdown to a bus
}).add;

SynthDef(\big, {|out|
    var sig = FSinOsc.ar(440!128); // generate 128 Fast Sine Oscillators
    Out.ar(out, sig)               // write a one-channel mixdown to a bus
}).add;
```

Listing 5.1: Synth used for benchmarks

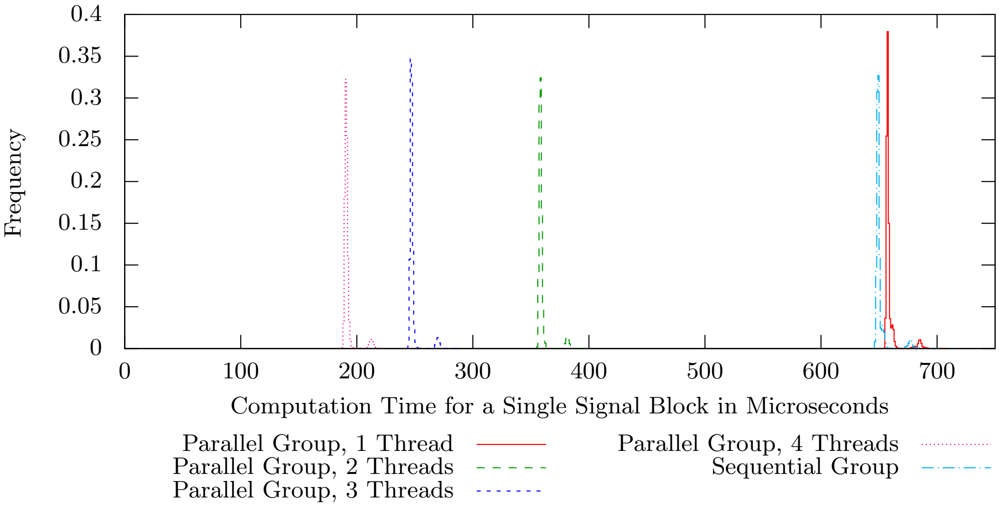

Figure 5.1: Execution Time Histogram, 256 Lightweight Synths (No Contention)

FSinOsc ugens, the heavyweight synth uses 128. The outputs of the FSinOsc ugens are mixed to a single channel and written to a bus which can be specified as synth control argument.

5.2.1 Single Parallel Group

For the first benchmark, 256 lightweight synths are run in a single parallel group. In this use case, the engine is mainly stressed when nodes are removed from the scheduler queue. This benchmark was run with two different settings. The results shown in Figure 5.1 are obtained when each synth is writing to a separate bus, while Figure 5.2 shows the results when writing to the same bus, which increases the contention. In this use case, the increased contention does not show a significant effect, though. While the measured difference of worst-case execution times has a maximum of 3 microseconds, the average-case execution time is even slightly better for the high-contention use case. This is probably because of better memory locality.

Figure 5.3 shows the result of a similar use case, but with 16 heavyweight synths instead of 256 lightweight ones.

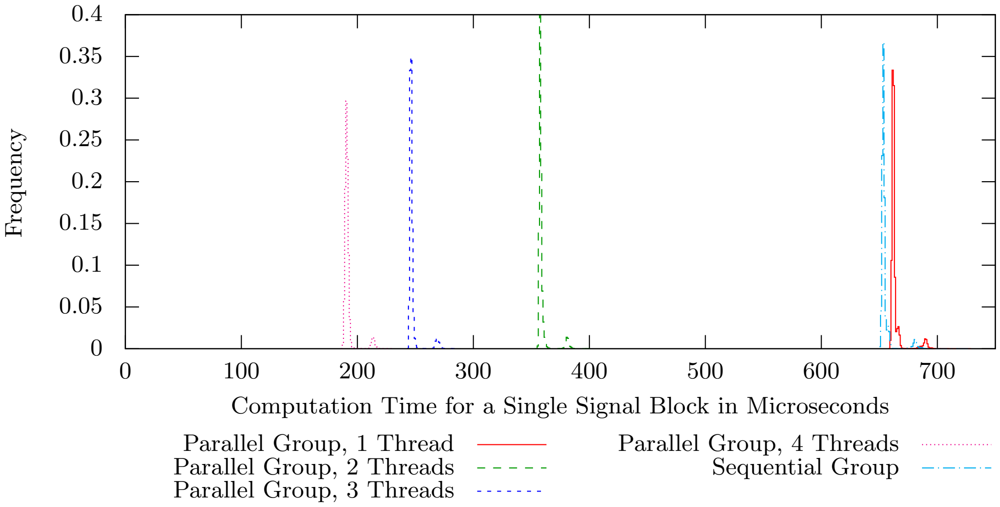

Figure 5.2: Execution Time Histogram, 256 Lightweight Synths (High Contention)

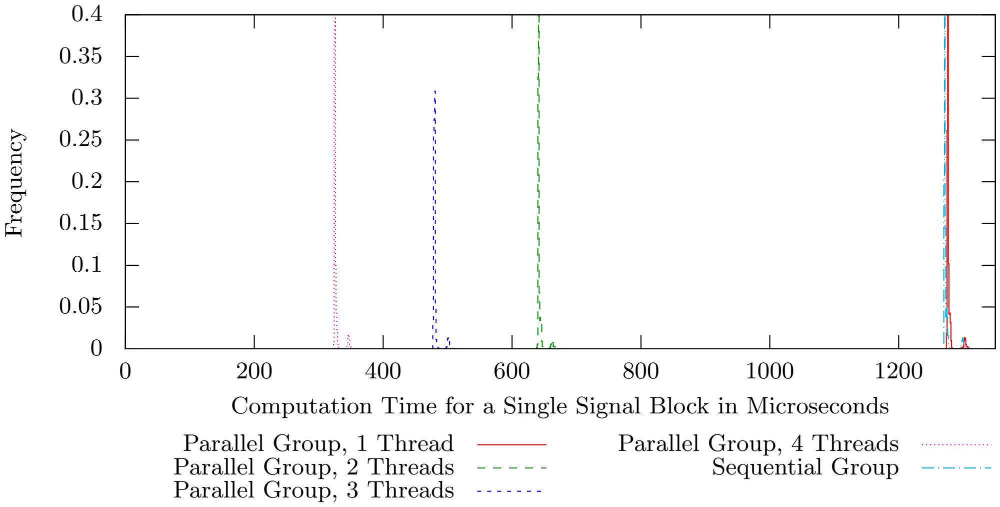

Figure 5.3: 16 Heavyweight Synths

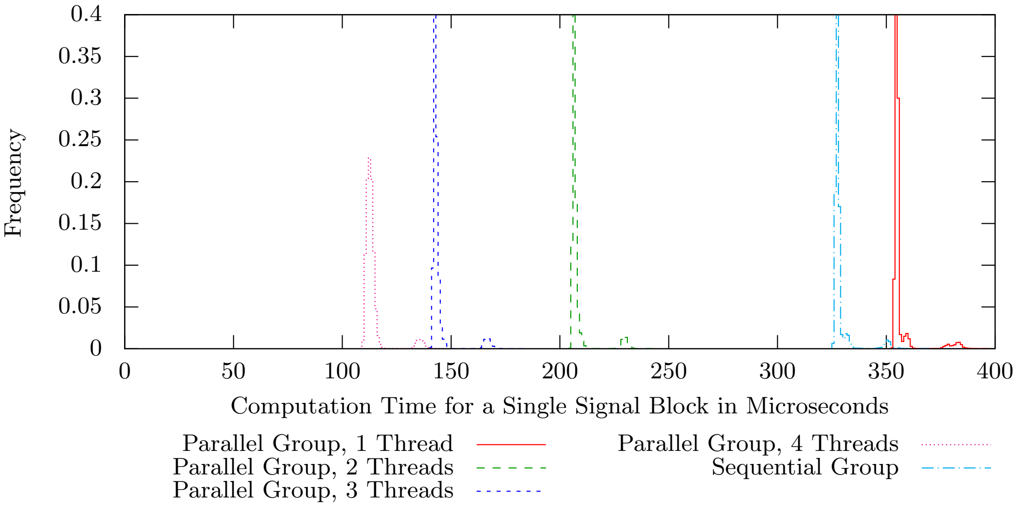

Figure 5.4: Execution Time Histogram, 2 Parallel Groups with Lightweight Synths

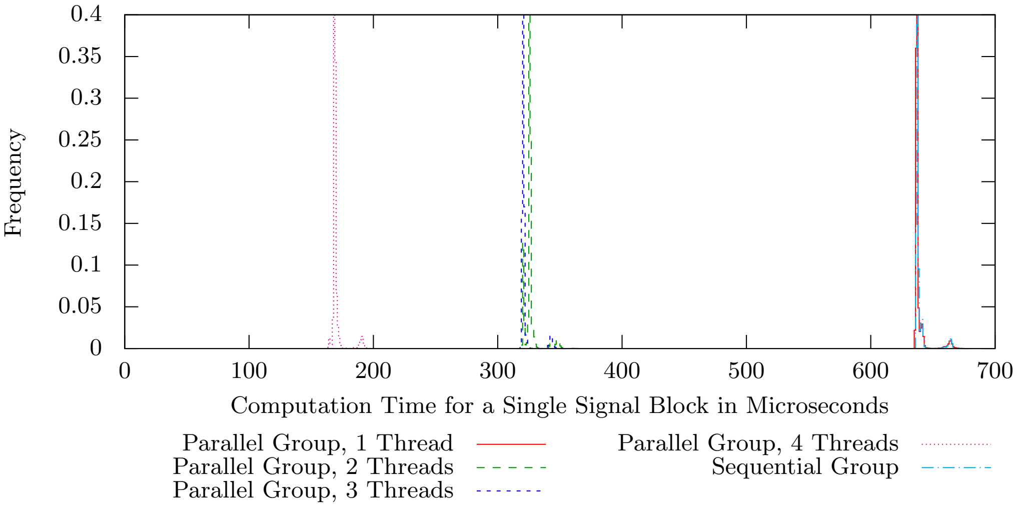

Figure 5.5: Execution Time Histogram, 2 Parallel Groups with Heavyweight Synths

5.2.2 Sequential Parallel Groups

To measure the performance impact of group traversal, a second use case was evaluated, based on two sequential parallel groups. In the first case, each parallel group contained 64 small synths, in the second case 4 big synths. The benchmark results can be seen in Figures 5.4 and 5.5.

These test cases stress a different part of the Supernova engine than the earlier test, since the node activation will add some non-trivial overhead. The node activation algorithm requires two steps. The first step is decreasing the activation count, the second is adding the node to the scheduler queue. With two sequential parallel groups each element of the preceeding group needs to decrease the activation count of each element of the following group. The algorithmic complexity is therefore quadratic. This explains the performance difference between sequential and parallel groups that is seen in the test case with lightweight synths. This effect is fortified in this benchmark, because all nodes have the same CPU costs, so the access to the activation counts occurs roughly at the same time. The contention will increase the effect of this issue.

In the case of heavyweight synths, the node activation overhead can be neglected. However the histogram shows a different interesting effect: when running the engine with 3 audio threads, the performance is similar than when running on 2 audio threads. This can be explained when the distribution of synths to the worker threads is considered. When 4 synths are distributed to three groups, first all 3 threads get some work to perform, but after these three synths have been evaluated, the fourth synth needs to be evaluated, before any other synth will be scheduled. During this time, the 'idle' threads do not perform any work, but busy-wait until there is more work for them to perform.

While these two cases are perhaps borderline cases, they show some implications that the user has to deal with when using Supernova.

5.2.3 Speedup & Discussion

The speedup, that one can achieve with parallel groups depends heavily on the use case. If the dispatching overhead is kept low, nearly a linear speedup can be achieved. However one can clearly see how some aspects reduce the scalability:

Node Scheduling

When using many small synthesis nodes, the dispatching overhead and scheduler queue contention adds some measurable overhead. The effect can be seen in the two test cases of small synths running in parallel. While both perform equally well, the test cases with big synths scale way better. One possible improvement would be to use work-stealing scheduler, as proposed by Letz et al [LOF10]. Combining nodes inside a parallel group may be introduce some further problems, since the performance cost per synthesis node is neither known nor can it assumed to be constant. While a heuristic could be used to improve the average-case performance, it is unlikely to improve the worst-case behavior.

Parallel Group Scheduling

Since the node graph is transformed to an internal dependency representation, some borderline cases are rather inefficient. The node scheduling of parallel groups is linear in the number of scheduling nodes and linear in the number of scheduled nodes. Therefore sequential parallel groups impose quite some overhead, that can clearly be observed. This behavior could be changed from a quadratic to a linear complexity by introducing a barrier node in between both parallel groups. While adding some constant overhead, this would probably increase the scalability.

Work Distribution

In order to avoid busy waiting and therefore lost CPU cycles, the scheduler queue needs to be filled as well as possible. This can be done, if the signal graph is formulated in a way that tries to keep the scheduler queue filled. The proposed extension of free node predecessors and successors as described in section 3.4 would make it easier than the current version using parallel groups.

When examining the results for the worst-case that are shown Figure 5.7, the scalability is worse than for the average case (Figure 5.6), although there are no substantial differences. The worst-case execution time is roughly 20 microseconds larger than the average. As this is not only the case for the multi-threaded, but also for the single-threaded benchmarks, this is probably due to hardware effects like NMIs.

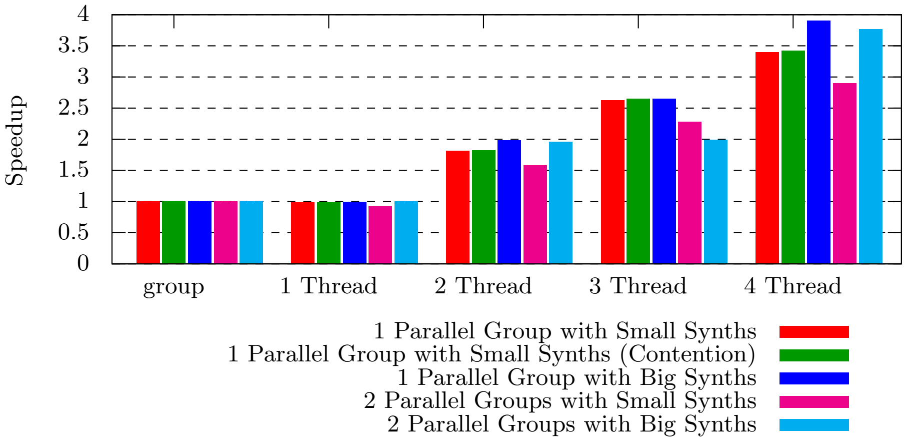

Figure 5.6: Average Case Speedup

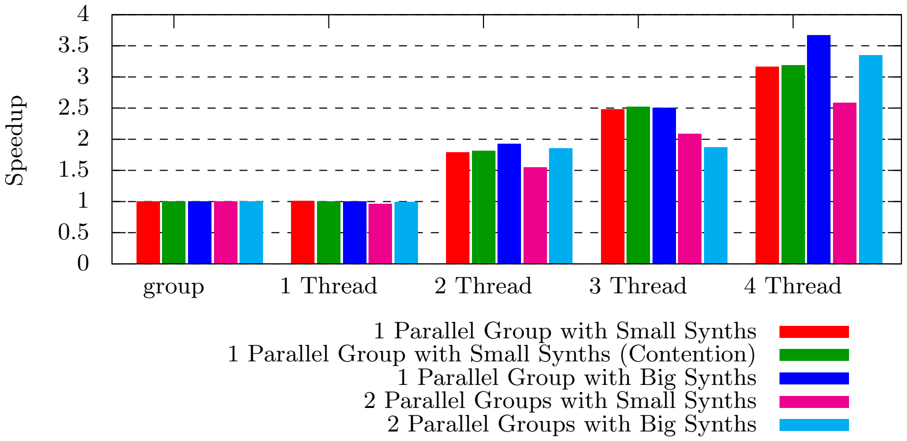

Figure 5.7: Worst Case Speedup

5.3 Benchmarking Latency Hotspots

The second part of the benchmarks focuses on the latency hotspots of the Supernova engine. The latency hotspots occur during the evaluation of asynchronous events, which can be triggered from received or scheduled OSC commands. Analyzing the source code for the event handlers suggests that the following operations are likely to introduce latency hotspots:

- synth creation
- node graph manipulation
- dsp queue creation

In order to evaluate the effects of these issues, a slightly different benchmarking setup was chosen. A parallel group with 32 small synths is followed by parallel group with 4 big synths. In parallel to this, a group of 4 small synths is followed by a group of 2 big synths. This setup

```
fork {
    var root = ParGroup.new;

    var section1 = Group.head(root);
    var section2 = Group.head(root);

    var section1_head = ParGroup.head(section1);
    var section1_tail = ParGroup.tail(section1);

    var section2_head = Group.head(section2);
    var section2_tail = Group.tail(section2);

    32.do { Synth.head(section1_head, \small); 0.1.wait };
    4.do  { Synth.head(section1_tail, \big);   0.1.wait };

    4.do  { Synth.head(section2_head, \small); 0.1.wait };
    2.do  { Synth.head(section2_tail, \big);   0.1.wait };
}
```

Listing 5.2: Graph layout for latency hotspots

increases the load to the dsp queue generation, since it adds some further complexity to the node graph. The actual code is shown in Listing 5.2. In order to increase the effect of DSP queue generation, the synth creation messages are throttled to ensure that for every node graph layout the corresponding DSP queue is actually generated.

5.3.1 Latency of Synth Instantiations

Examining the measured execution times for synth creation, one can clearly see that it is actually governed by the complexity of the synth. the lightweight synth with just a few unit generators is faster to instantiate than the heavyweight synth. The costs of instantiating a synths are on the order of tens of microseconds. The functionality is the same in Supernova and in scsynth, so scsynth would probably show a similar behavior. Measurements suggest synth instantiation time of 10 microseconds for the 'small' synth and of 30 microseconds for the 'big' synth.

5.3.2 Latency of Node Graph Manipulations

Node graph manipulations are a bit different in Supernova and in scsynth. While scsynth organizes the synthesis nodes in nested linked lists, Supernova maintains a graph structure. The costs for maintaining the graph structure are barely measurable for this use case, in most cases the results are below one microsecond, with a measured worse-case of 3 microseconds.

5.3.3 Latency of Dsp Queue Creation

The generation of the dsp queue is the only latency hotspot in Supernova that has no equivalent in scsynth. The implementation guarantees that it does not happen more than once per control rate period, though. In the examined use case, the average-case CPU costs are about 4 microseconds with a worst-case bound of 13 microseconds. While this scales linearly with the size of the node graph, the costs are still reasonably low.

5.3.4 Discussion

The additional latency hotspots for maintaining the node graph and creating the DSP queue do not seem to have a significant impact on the overall scalability of the Supernova engine. While being measurable, the costs are reasonably low so they will probably not be significant in real-world applications.

Chapter 6

Related Work

The use of parallelism in computer music systems dates back to the late 1980s and early 1990s, when audio synthesis was a highly demanding application for the available CPU power. In those days, some approaches have been pioneered, although the available hardware and synchronization primitives were less powerful back then. Today we face a different situation. The floating-point performance of current CPUs is sufficient for many computer music tasks, however the continuous growth of single-processor performance is coming to an end because of physical constraints in the manufacturing process of microprocessors [Kis02]. To continuously increase the CPU speed, processor vendors start focusing on parallel architectures, integrating multiple CPU cores into one processor.

6.1 Max-like languages

Max-like languages [Puc02] are a family of visual programming environments, using a data-flow graph to represent both control data and audio signals, which shares some analogy to connecting modules of an analogue synthesizer. There is no formal definition of the 'max language', but most of the systems share the same concepts, although the library of objects varies.

The first max-like language was **Patcher**, a system developed by Miller Puckette at IRCAM (Institut de Recherche et Coordination Acoustique/Musique, Paris) in 1986 for the realization of Philippe Manoury's composition "Pluton" [Puc88]. Patcher was able to do MIDI event processing and to control external audio synthesis hardware. Patcher was rewritten in C and called Max (named after Max Mathews) during the following year.

IRCAM started to develop the IRCAM Signal Processing Workstation (ISPW) in 1989, a NeXT computer with an extension board of 2 to 24 Intel i860 coprocessors for real-time signal processing [Puc91b], running the CPOS operating system, which was specifically written for the real-time requirements of real-time audio synthesis [Puc91b]. It was the first computer music system that was built with general-purpose processors, while earlier systems like the Samson Box [III91] or the 4X [Sch] used special synthesis hardware. The ISPW was running Max as user interface and control engine for the real-time signal processing engine **FTS** (Faster Than Sound). FTS could be controlled by special Max objects, called **tilde objects** [Puc91a].

In 1996, Miller Puckette started **Pure Data (Pd)**, a new implementation of the original Max, which was released as open source software under the BSD license [Puc96]. In 1997, Cycling 74, a San Francisco based company, founded by former IRCAM employee David Zincarelli, extended Max with a signal processing engine based on Pure Data and released the proprietary Software **Max/MSP**, which is available on the Windows and Mac OSX platforms and got widely used both in computer music and other fields of computer-based arts.

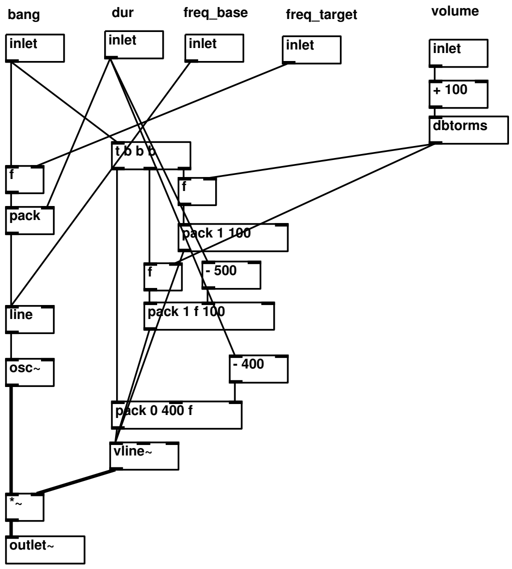

Figure 6.1: Pd abstraction for sine sweep

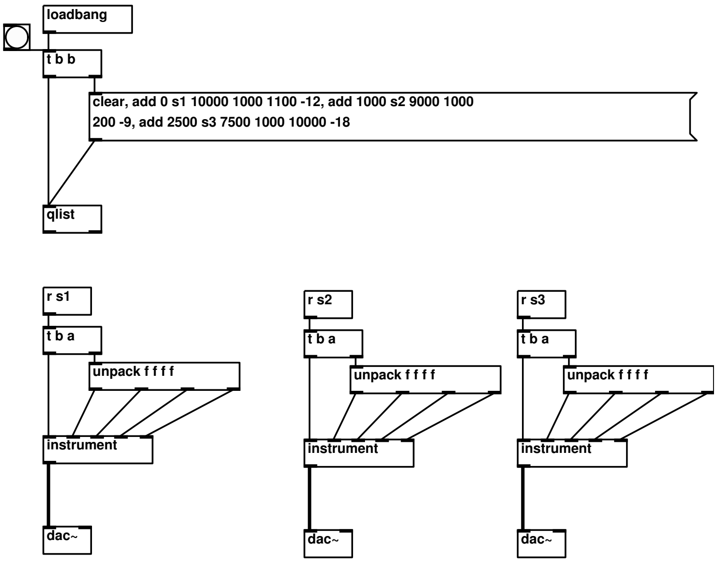

Figure 6.2: 3 sweeping sine waves with a Pd patch

The “Real Time Systems” group at IRCAM, led by François Déchelle, reengineered the architecture of Max/FTS, splitting the engine from the graphical user interface, which was written from scratch using the Java programming language, and released the new program under the name **jMax** [DBdC^{+}98]. It was licensed as GNU Lesser Public License, running under Linux, Mac OSX and Windows, but never got as widely used as Max/MSP or Pure Data. In 2004 the IRCAM stopped the development of jMax, although it was revived as a community project by two former developers in 2008 [dCDM09].

Programs written in max-like languages are called **patches**. Patches are written in a visual programming environment, where graphical objects can be placed on a canvas. These objects can be interconnected, in order to form the data-flow graphs for audio signals and control messages. Objects can be written in the max language itself. Patches that are stored in the search path, can be used as objects in another patch. These patches are called **abstractions**. Figure 6.1 shows an abstraction written in Pure Data to play a sine sweep. The signal from the output of the sine wave oscillator `osc~` is multiplied with an envelope, generated by the `vline~` object. Figure 6.2 shows, how to use this abstraction in a patch. The `inlet` and `outlet~` objects inside the abstraction refer to inlets and outlets of the instantiated abstraction in the parent patch. The trailing ~ is a convention to denote an audio object. In order to play 3 sounds at the same time, the abstraction needs to be instantiated once for each sound. The `qlist` object serves as message sequencer and sending lists to certain receivers at a given time.

6.1.1 Audio Synthesis in Max-like Languages

Max-like languages have no real equivalent to synthdefs and synths. Instead, abstractions can be used to encapsulate the audio synthesis to small entities. Most systems[1] contain a single ugen graph, that is used by all patches of a running system. When an abstraction is instantiated, its tilde objects will be added to this graph and the graph will need to be traversed again to build a sequential list and allocate signal buffers. This adds some significant overhead, which makes it impossible to change the signal graph in real-time without experiencing audio dropouts. The example is shown using Pure Data, where abstractions need to be instantiated and voices need to be allocated manually.

Using the terminology introduced in Section 2.2, signal graphs in max-like languages model a fine-grained graph. While in theory, they could be parallelized, implicit dependencies introduce practical problems, making the parallelization hard, if not impossible to implement. Instead, these systems introduce extensions, providing multi-processor support with some limitations.

6.1.2 FTS

FTS was designed to run on the parallel hardware of the ISPW. The multiprocessor capability is exposed explicitly to the user, who can assign all tilde objects of a window to run on a certain processor [Puc91a]. Internally this is achieved by using the pipelining technique described in Section 2.3. If signals are passed between processors, a latency of one signal block (typically 64 samples) is introduced. Since the processors are assigned statically, an application would have to be optimized for a certain number of CPUs.

6.1.3 Max/MSP: poly~

Version 5 of Max/MSP, that has been released in 2008, provides a `poly~` object, which makes it possible to run different instances of one abstraction in separate threads. This way it is only possible to run instances of the same patch in parallel. With this approach it is not possible to run multiple different abstractions in parallel. Unfortunately it is not known, if the `poly~` object uses pipelining techniques, introducing additional latency, nor how resource access is managed.

The `poly~` object does not introduce a general solution, but can just provide parallelism for a special use case.

[1] This applies to Pure Data, that I have studied in great detail. It also seems to apply to other max-like systems, though.

```
                    ; define global variables:
sr     = 44100      ; audio sample rate
kr     = 4410       ; control sample rate
ksmps  = 10         ; audio samples per control rate
nchnls = 1          ; number of output channels

instr 1
 idur          = p3
 ifreq_base    = p4
 ifreq_target  = p5
 iamp          = ampdbfs(p6)

 k_sweep       line ifreq_base, idur, ifreq_target ; frequency
 a_sine        oscil iamp, k_sweep, 1              ; sine oscillator
 k_env         linen 1, 0.1, idur, 0.4             ; envelope
               out a_sine * k_env                  ; output signal
endin
```

Listing 6.1: Csound Instrument for sine sweep

6.1.4 Pure Data: pd~

Recent versions of Pure Data provide a `pd~` object, that implements a concept similar to the approach of FTS. The `pd~` creates a subpatch, which starts a separate thread in the background [Puc08]. When data is passed across a `pd~` boundary, a latency of one signal block is introduced, that is common to all implementations using the pipelining technique. Apart from the additional latency, this approach does not scale well on current computer systems. Since each `pd~` object instantiates a separate process, the worst-case latency for the main Pd process to finish grows linearly with the number of `pd~` instances. This limits its usability when running in low-latency environments on untuned systems (compare Section 4.1.1). Since child processes are created via the `fork` system call, duplicating the main Pd process, page-fault related latencies may occur, since the child process does not inherit the parent's memory locks and writing to copy-on-write pages will cause minor page faults (compare Section 4.1.2).

6.2 Csound

The only "Music N" based language which is still widely used is Csound [Ver07], which was developed at MIT by Barry L. Vercoe as a successor of Music 360, a "Music N" programm for IBM 360 computers. It was implemented using the C programming language to increase the portability. While earlier "Music N" systems were designed as a compiler for sound files, Real-time Csound, presented in 1990 at ICMC Glasgow [VE90], also was able to generate sounds in real-time. Csound is currently maintained by John ffitch at the University of Bath and is still widely used in academic computer music. It became quite common to use it not only as standalone application, but also with gui front-ends, as LADSPA [LW07] or VST plug-in [Ver07] or embedded in systems like Max/MSP or Pure Data. Csound is free software, released under the GNU Lesser General Public License.

In Csound, score and orchestra are written independently, either as two separate files or as one "Unified Csound File", using XML-like tags to separate different code sections. The orchestra file is based on a simple declarative syntax, and defines audio-rate, control-rate and number of

```
f1 0   8192   10     1       ; fill table with sine wave
i1 0     10 1000  1100   -12 ; play 3 sweeping notes
i1 1      9 1000   200   -9
i1 2.5  7.5 1000 10000  -18
```

Listing 6.2: Csound score for playing 3 notes

channels for the resulting output and a list of instruments. An instrument for playing a sweeping sine (similar to the synthdef in Section 1.2.2 can be found in Listing 6.1. The first letter of instrument variables are used to specify the rate of the opcode: `a` denotes audio-rate, `k` control rate and `i` instrument rate.

The score language is based on a list of statements for the generation of wavetables for wavetable oscillators (F-Statements) and a note list, specifying a note based on instrument number, start time, note duration and a number of instrument arguments (I-Statements) [Bou00]. Listing 6.2 shows how to write a score with 3 notes and one wave table. Since the score language is just a list of definitions, its expressive power for algorithmic composition is quite limited. However it can serve as intermediate language for score descriptions. It is a common technique to generate Csound scores from other systems for computer assisted compostition like OpenMusic [ARL+99].

Instruments can share data by using global variables, f-tables (audio buffers) or by using the so-called ‘zak’ bus system.

6.2.1 Parallelizing Csound

According to John ffitch, there have been several attempts to parallelize Csound [fDB09]. The main difficulty when parallelizing Csound is to maintain the semantics of the language. The first attempt of a parallel implementation of Csound dates back to 1989, when an array of about 170 Transputers was targeted [fDB09].

Csound’s original developer Barry Vercoe implemented an “Extended Csound”, targeting SHARC DSP processors, which didn’t strictly follow the semantics of the Csound language, since it didn’t support communication between instruments via global variables with the exception of passing the instrument outputs to a single effect instrument and passing the output of the effect instrument to a reverberation instrument [ffi09].

In 1997, John Williams and Mark Clement published a paper describing a distributed implementation of Csound. The Csound program was split into a server and a client application, which is able to split the score file to separate notes and distribute each note to available servers on the local network [WC97].

Recently, some work has been done to implement a parallel version on Csound 5 [ffi09]. It follows the idea to create a dependency graph between instruments, in order to maintain the semantics of the language. In Csound, this is somehow easier to achieve than in SuperCollider, since the instrument parser has some knowledge about global variables and opcodes. When the instrument section is parsed, the instruments are annotated by synchronization constraints, like read or write access to global variables, f-tables or the ‘zak’ bus. In cases where it cannot be fully determined, instruments are marked as such.

Based on this analysis instruments can be identified, that can be computed in parallel while keeping the correct semantics. Internally, it uses a database of unit generator costs to estimate the run-time costs of instruments, so that low-overhead instruments can be clustered and scheduled as one entity to avoid scheduling overheads. This database provides cost values in CPU cycles for

UGen initialization, costs per control cycle and costs per audio sample. It shares some analogy to the PTI, proposed by Reiter and Partzsch (compare Section 2.2.2).

Unfortunately, neither ffitch [ffi09] nor Wilson [Wil09] provides a real-time analysis of its engine. Wilson provides several benchmarks, but they focus on throughput for non real-time synthesis. In his real-world benchmarks (comparing 5 different scores), only 2 show a significant speedup when using 2 CPUs instead of one. Using 4 CPUs is always less efficient than using 3 CPUs, which suggests a certain limitation either in the instrument scheduler or the automatic parallelization. Wilson briefly discusses the real-time performance, but without providing any quantitative results. ffitch [ffi11] provides some benchmark results of the parallel implementation with up to 5 threads. His benchmarks only show modest performance gains: while in some cases the parallel implementation is slightly more efficient than the sequential one, using more threads is not necessarily more efficient. The best speedup of about 1.3 is achieved with 3 threads, in many cases (especially for small block sizes) the parallel implementation is even slower than the sequential one.

While one cannot compare the real-world benchmarks of ffitch and Wilson with the synthetic benchmarks of the Supernova engine from Chapter 5, one can still assume that the Supernova engine is way more scalable than the engine of the current version of parallel csound. ffitch admits that the performance results of csound are “a little disappointing” [ffi11] and suggests to use higher control rates to be able to get better speedup. But as increasing the control rates will also increase the audibility of zipper noise[2] and decrease the time resolution of the audio synthesis, this may introduce some further issues.

6.3 Distributed Engines

Several programs are designed to be used in parallel on distributed systems. Since SuperCollider has a strict separation between language and synthesis engine communicating via OSC, it is possible to control multiple server instances, possibly running on different computers of a network, from one single sclang client. This is commonly used for many computationally intensive applications like wave field synthesis. The 192 channel wave field synthesis system of the Game of Life foundation in Leiden/Netherlands [Negon] is completely written in SuperCollider is using 2 8-core computers, each of them running one 8 instances of scsynth.

The 832 channel WFS system in the large lecture hall at TU Berlin is driven by the distributed software Wonder [Baa07]. Wonder is a software system of different components, a control unit, a score playback/recording engine and an audio rendering system for direct rendering of virtual audio sources and for room simulation. The system consists of one control computer, that is controlling a cluster of 15 linux computers, each controlling 56 loudspeakers. The communication is realized using the OSC protocol.

6.4 Faust

Faust is a functional programming language for block diagram composition. Its programs are textual definitions of signal processing block diagrams. Faust does not work as standalone program, but is used as code generator, compiling the block diagrams to C++ classes. The Faust distribution provides wrappers to directly use generated classes as with Jack or ALSA audio backend and Gtk GUI, as VST or LADSPA plugins, or as objects in Max/MSP, Pure Data or SuperCollider [GKO06].

[2] The reason for zipper noise is the quantization of control rate signals

The Faust compiler includes optimizations for automatic vectorization and parallelization. The vectorized C++ code is a reorganization of the original C++ code, based on loop unrolling techniques in order to help the automatic vectorizer of the compiler (see Chapter 2.1.1). The parallel code generator was originally implemented using the OpenMP API [OLF09], which has recently been replaced with a custom work-stealing scheduler [LOF10].

According to Orlarey et al, the performance of the OpenMP backend depends heavily on the implementation. The work-stealing scheduler provides better performance due to reduced contention. Depending on the type of the dsp algorithm, Orlarey's benchmarks show some speedups, but also some performance loss. Unfortunately, the benchmarks focus on throughput. It would be interesting to see worst-case benchmarks of Faust-generated code.

Some personal discussions with Orlarey and Letz showed, that it may not necessarily be the best approach for Supernova. The node graphs of Faust and supernova differ in a few ways. Faust node graphs are fine grained, while Supernova node graphs are coarse grained, so the overhead that is caused by the scheduler queue contention is a bigger issue in Faust. On the other hand, the structure of typical Supernova node graph will probably cause the scheduler jobs to be distributed poorly among the threads.

6.5 Occam

The idea of specifying parallel and sequential parts of a program explicitly has quite some history in computer science. The concurrent programming language occam [Hyd95] supports the keywords **SEQ** and **PAR**, that are used to specify statements, that should be evaluated sequentially or parallel. occam implements the Communicating Sequential Processes (CSP) concept. The actual communication between processes is realized via a message passing interface using channels. Occam was introduced in the early 1980s by Inmos Limited, who eventually designed the Transputer, a parallel computer system, designed for the CSP concurrency model. This syntax influenced some other implementations of the CSP like the **parallel** keyword in SuperPascal [Han94] or Java-style **par** blocks in ProcessJ [Ped].

SEQ and **PAR** style syntax shares some analogies with groups and parallel groups. While they specify how statements in a block are supposed to be evaluated, groups and parallel groups define how their child nodes will be executed. However the syntax is not related at all and parallel groups do not implement the CSP.

Chapter 7

Conclusion & Future Work

7.1 Conclusion

The combination of SuperCollider with Supernova is the first computer music systems that makes use of multiprocessor hardware in a generic and scalable way. Since late 2009, I have personally used it in more than 20 concerts.

Since Supernova is a drop-in replacement for the SuperCollider server scsynth, it can smoothly be integrated into existing system. After some discussions during the SuperCollider Symposium 2010, the SuperCollider developers agreed to integrate Supernova into the next 3.5 release as an alternative to scsynth and eventually to completely replace scsynth with Supernova. As a first step, the required extensions to the unit generator API have already been merged into the SuperCollider repository.

The parallel audio engine of Supernova provides a combination of features that cannot be found in any other system:

No Additional Latency
Since Supernova does not rely on pipelining techniques, it does not add any latency to the audio signal.

Explicit Handling of Parallelism
Supernova does not try to parallelize the existing signal graph automatically, but exposes the parallelism to the user. This greatly simplifies the implementation, since no resource-based dependency analysis needs to be executed and avoids the limitation of a dependency analysis, which would reduce parallelism if the resource access pattern could not be predicted correctly.

Real-Time Safety
Supernova is optimized for real-time operations. While many other parallel computer music systems seem to focus on throughput, Supernova is optimized for the worst-case behavior. Benchmarks suggest that the worst-case behavior is mainly governed by hardware and operating systems.

Scalability
Supernova is designed for dynamically changing signal graphs and can adapt to the number of available CPUs. Benchmarks suggest that a considerable speedup can be achieved, if the user can parallelize his program such that the scheduler queue always contains some work. Some synthetic benchmarks even show a linear speedup.

Supernova is completely backwards compatible with existing SuperCollider code. Code written for SuperCollider will run under Supernova as it would under scsynth. While it won't benefit from parallel hardware if run under scsynth, it won't suffer any performance impact, either. The extension of 'parallel groups', that is implemented in Supernova can be easily emulated with facilities from scsynth, making the extension also forward compatible. The proposed 'satellite nodes' will not be compatible with scsynth.

The Nova.SIMD framework provides a generic way to utilize data-level parallelism. It provides an abstract interface that is implemented with different SIMD instruction sets, and provides many algorithms that are commonly found in computer music systems. While it is not a part of Supernova itself, it has been included into the SuperCollider distribution in order to speed up several unit generators.

7.2 Future Work

The implementation of Supernova raised some interesting questions.

DSP Queue Caching

Currently, the dsp queue is completely regenerated whenever the node graph is changed. Therefore, some optimizations are possible, reducing the worst-case execution time of the queue generation algorithm. In many cases, some parts of the dsp queue could be cached and later reused. This would probably affect the worst-case behavior, but may improve the performance in the average case. Depending on the use case, this could reduce the probability of queue-generation related audio dropouts.

JIT Compilation of Synthdefs

All current computer music systems build a unit generator graph, which is dispatched by calling the ugen functions indirectly using function pointers. It would be interesting to compile synthdefs to native code using JIT compilation techniques, to enable compiler optimizations that could work across ugen boundaries. As cross-ugen optimizations would reduce the required memory bandwidth if local data can be stored in registers, this may increase multiprocessor scalability.

Synchronous Sclang

The current design of SuperCollider is based on the strong separation between audio synthesis and control language. This design has been introduced in the transition from the commercial SuperCollider 2 to the open source SuperCollider 3. Before that, it was possible to run the language interpreter synchronous to the audio synthesis. It would be interesting to re-introduce the process separation by integrating sclang back into the parallel architecture of Supernova and introduce a facility to run sclang code synchronous to the audio signal processing.

While these issues are outside the scope of this master's thesis, they offer some interesting topics for further research.

Bibliography

[ARL$^+$99] Gérard Assayag, Camilo Rueda, Mikael Laurson, Carlos Agon, and Olivier Delerue. Computer Assisted Composition at Ircam: PatchWork & OpenMusic. *Computer Music Journal*, 23(3), 1999.

[Baa07] Marije A. J. Baalman. *On Wave Field Synthesis and electro-acoustic music, with a particular focus on the reproduction of arbitrarily shaped sound sources.* PhD thesis, Technische Universität Berlin, 2007.

[Ble08] Tim Blechmann. nova - A New Computer Music System with a Dataflow Syntax. Technical report, 2008.

[BMBW00] Emery Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. *ACM SIGPLAN Notices*, 35(11):117–128, 2000.

[Boh09] Remy Bohmer. HOWTO: Build an RT-application. `http://rt.wiki.kernel.org/index.php/HOWTO:_Build_an_RT-application`, 2009. [Online, Accessed November 4th, 2009].

[Bou00] Richard Boulanger, editor. *The Csound Book: Perspectives in Software Synthesis, Sound Design, Signal Processing,and Programming.* The MIT Press, March 2000.

[CJL19] Sylvain Chevillard, Miaora Joldeş, and Christoph Lauter. Sollya: An Environment for the Development of Numerical Codes. *Mathematical Software–ICMS 2010*, pages 28–31, 2919.

[Dan08] Roger B. Dannenberg. Is Music Audio Processing Embarrassingly Parallel? In *Proceedings of the International Computer Music Conference*, 2008.

[DBdC$^+$98] François Déchelle, Riccardo Borghesi, Maurizio de Cecco, Enzo Maggi, Joseph B. Rovan, and Norbert Schnell. Latest evolutions of the FTS real-time engine: typing, scoping, threading, compiling. In *Proceedings of the International Computer Music Conference*, 1998.

[dCDM09] Maurizio de Cecco, François Déchelle, and Enzo Maggi. jMax Phoenix: le jMax nouveau est arrivé. In *Proceedings of the Linux Audio Conference*, 2009.

[Der08] Helmut Dersch. Universal SIMD-Mathlibrary. Technical report, Furtwangen University of Applied Sciences, August 2008.

[DG02] David Dice and Alex Garthwaite. Mostly lock-free malloc. In *Proceedings of the 3rd international symposium on Memory management*, page 174. ACM, 2002.

[Dun92] David Dunn. A History of Electronic Music Pioneers. *ders.(Hrsg.), Eigenwelt der Apparate-Welt.(Katalog), Linz*, pages 21–62, 1992.

[fDB09] John ffitch, Richard Dobson, and Russell Bradford. The Imperative for High-Performance Audio Computing. In *Proceedings of the Linux Audio Conference*, pages 73–79, 2009.

[ffi09] John ffitch. Parallel Execution of Csound. In *Proceedings of the International Computer Music Conference*, 2009.

[ffi11] John ffitch. Running Csound in Parallel. In *Proceedings of the Linux Audio Conference 2011*, 2011.

[GKO06] Albert Gräf, Stefan Kersten, and Yann Orlarey. DSP Programming with Faust, Q and SuperCollider. In *Proceedings of the Linux Audio Conference*, 2006.

[GM] Sanjay Ghemawat and Paul Menage. TCMalloc: Thread-Caching Malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[Gog] Michael Gogins. Double Blind Listening Tests of Csound 5 Compiled with Single-Precision and Double-Precision Samples. http://ruccas.org/pub/Gogins/csoundabx.pdf.

[Gol06] Lois Goldthwaite. Technical report on C++ performance. *ISO/IEC PDTR*, 18015, 2006.

[GPT05] Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. Allocating Memory in a Lock-Free Manner. *Algorithms–Esa 2005: 13th Annual European Symposium, Palma de Mallorca, Spain, October 3-6, 2005: Proceedings*, 2005.

[Han94] Per Brinch Hansen. SuperPascal-a publication language for parallel scientific computing. *Concurrency: practice and experience*, 6(5):461–483, 1994.

[Hyd95] Daniel C. Hyde. Introduction to the programming language Occam. *Department of Computer Science Bucknell University, Lewisburg*, 1995.

[III91] Julius O. Smith III. Viewpoints on the History of Digital Synthesis. In *Proceedings of the International Computer Music Conference*, pages 1–10, 1991.

[Inta] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual.*

[Intb] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual.*

[Kis02] Laszlo B. Kish. End of Moore's law: thermal (noise) death of integration in micro and nano electronics. *Physics Letters A*, 305(3-4):144–149, 2002.

[LOF05] Stéphane Letz, Yann Orlarey, and Dominique Fober. Jack audio server for multiprocessor machines. In *Proceedings of the International Computer Music Conference*, 2005.

[LOF10] Stéphane Letz, Yann Orlarey, and Dominique Fober. Work Stealing Scheduler for Automatic Parallelization in Faust. In *Proceedings of the Linux Audio Conference*, 2010.

[LW07] Victor Lazzarini and Rory Walsh. Developing LADSPA plugins with Csound. In *Proceedings of the 5th International Linux Audio Conference*, pages 30–36, Technical University Berlin, 2007.

[M^{+}96] James McCartney et al. *SuperCollider Manual*, 1996.

[Mag06] Thor Magnusson. Affordances and constraints in screen-based musical instruments. In *NordiCHI '06: Proceedings of the 4th Nordic conference on Human-computer interaction*, pages 441–444, New York, NY, USA, 2006. ACM.

[Man04] Peter Manning. *Electronic and computer music.* Oxford University Press, USA, 2004.

[McC96] James McCartney. SuperCollider, a new real time synthesis language. In *Proceedings of the International Computer Music Conference*, 1996.

[McC02] James McCartney. Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal*, 26(4):61–68, 2002.

[McC09] James McCartney. Re: Re: supernova v0.1. sc-dev Mailing List, http://article.gmane.org/gmane.comp.audio.supercollider.devel/21059, 2009.

[Mic04] M. M. Michael. Scalable lock-free dynamic memory allocation. *ACM SIGPLAN Notices*, 39(6):35–46, 2004.

[MMM^{+}69] Max V. Mathews, Joan E. Miller, F. R. Moore, John R. Pierce, and J. C. Risset. *The Technology of Computer Music.* The MIT Press, 1969.

[Mos92] Steve Moshier. Cephes mathematical library. http://www.netlib.org/cephes/, 1992.

[MRCR04] Miguel Masmano, Ismael Ripoll, Alfons Crespo, and Jorge Real. TLSF: A New Dynamic Memory Allocator for Real-Time Systems. *Real-Time Systems, Euromicro Conference on*, 0:79–86, 2004.

[Nai04] Dorit Naishlos. Autovectorization in GCC. In *Proceedings of the 2004 GCC Developers Summit*, pages 105–118, 2004.

[Negon] Miguel Negrão. The challenges and possibilities of real-time Wave Field Synthesis. In *Proceedings of the SuperCollider Symposium*, Berlin, Germany, in preparation.

[OH05] Kunle Olukotun and Lance Hammond. The Future of Microprocessors. *Queue*, 3(7):26–29, 2005.

[OLF08] Yann Orlarey, Stéphane Letz, and Dominique Forber. Multicore Technologies in Jack and Faust. In *Proceedings of the International Computer Music Conference*, 2008.

[OLF09] Yann Orlarey, Stéphane Letz, and Dominique Fober. Adding Automatic Parallelization to Faust. In *Proceedings of the Linux Audio Conference*, 2009.

[Ped] Matt Pedersen. ProcessJ. http://www.egr.unlv.edu/~matt/research/Joccam-content.html. [Online, Accessed July 22th, 2010].

[Puc88] Miller Puckette. The patcher. In *Proceedings of the International Computer Music Conference*, pages 420–429, 1988.

[Puc91a] Miller Puckette. Combining Event and Signal Processing in the MAX Graphical Programming Environment. *Computer Music Journal*, 15(3):68–77, 1991.

[Puc91b] Miller Puckette. FTS: A Real-time Monitor for Multiprocessor Music Synthesis. *Computer Music Journal*, 15(3):58–67, 1991.

[Puc96] Miller Puckette. Pure Data: another integrated computer music environment. In *Proc. the Second Intercollege Computer Music Concerts*, pages 37–41, 1996.

[Puc02] Miller Puckette. Max at Seventeen. *Computer Music Journal*, 26(4):31–43, 2002.

[Puc08] Miller Puckette. Thoughts on Parallel Computing for Music. In *Proceedings of the International Computer Music Conference*, 2008.

[RP07] Ulrich Reiter and Andreas Partzsch. Multi Core / Multi Thread Processing in Object Based Real Time Audio Rendering: Approaches and Solutions for an Optimization Problem. In *Audio Engineering Society 122th Convention*, 2007.

[SAN06] Scott Schneider, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *Proceedings of the 5th international symposium on Memory management*, pages 84–94. ACM New York, NY, USA, 2006.

[Sch] Arie van Schutterhoef. Sogitec 4X. http://knorretje.hku.nl/wiki/Sogitec_4X. [Online, Accessed October 4th, 2009].

[Sup97] Martin Supper. *Elektroakustische Musik und Computermusik: Geschichte, Ästhetik, Methoden, Systeme*. Wolke, 1997.

[VE90] Barry Vercoe and Dan Ellis. Real-Time CSOUND: Software Synthesis with Sensing and Control. In *Proceedings of the International Computer Music Conference*, pages 209–211, 1990.

[Ver07] Barry Vercoe. *The Canonical Csound Reference Manual Version 5.07*, 2007.

[WC97] John Williams and Mark J. Clement. Distributed polyphonic music synthesis. In *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing*, pages 20–29, 1997.

[Wes08] David Wessel. Reinventing Audio and Music Computation for Many-Core Processors. In *Proceedings of the International Computer Music Conference*, 2008.

[WF97] Matt Wright and Adrian Freed. Open Sound Control: A New Protocol for Communicating with Sound Synthesizers. In *In Proceedings of the 1997 International Computer Music Conference*, pages 101–104, 1997.

[Wil09] Christopher Wilson. Csound Parallelism. Technical Report CSBU-2009-07, Department of Computer Science, University of Bath, 2009.

[WJ93] Paul R Wilson and Mark S. Johnstone. Real-time non-copying garbage collection. In *ACM OOPSLA Workshop on Memory Management and Garbage Collection*, 1993.

[Xen01] Iannis Xenakis. *Formalized Music: Thought and Mathematics in Composition (Harmonologia Series, No 6)*. Pendragon Pr, 2nd edition, March 2001.