Multicore technologies in Jack and Faust

Y. Orlarey, S. Letz, D. Fober
Grame

ICMC’08, Belfast, August 2008
Outline

1. Jack

2. Faust
   - Overview
   - Parallel code generation
   - Performances

3. Conclusion
Jack Audio Server

Jack is a low latency audio server that runs on Linux, Macosx and Windows
The first versions of Jack were based on a sequential activation mechanism finely tuned for mono-core machines, but unable to take advantage of modern multi-core machines. A “topological sort” was used to find an activation order (A, B, C, D or B, A, C, D here).

Original Jack Activation Model

- Input
- Driver
- Client
- Output

Diagram:

- A
- B
- C
- D

Activation Order: A → C → D
The first versions of Jack were based on a sequential activation mechanism finely tuned for mono-core machines, but unable to take advantage of modern multi-core machines.
The first versions of Jack were based on a sequential activation mechanism finely tuned for mono-core machines, but unable to take advantage of modern multi-core machines. A “topological sort” was used to find an activation order (A, B, C, D or B, A, C, D here)
In the new semi-dataflow model an application in the graph becomes *runnable* when all inputs are available. Each client uses an *activation counter* to count the number of input clients which it depends on.
In the new semi-dataflow model an application in the graph becomes *runnable* when all inputs are available. Each client uses an *activation counter* to count the number of input clients which it depends on.
Semi-Dataflow Activation Model in action

State 1

State 2

CPU1

CPU2

State 3

State 4
Various types of activation

Jack proposes various types of activations
Various types of activation

Jack proposes various types of activations

1. **Synchronous**
Various types of activation

Jack proposes various types of activations

1. *Synchronous*
2. *Asynchronous*
Various types of activation

Jack proposes various types of activations

1. *Synchronous*
2. *Asynchronous*
3. *Free-wheel*
Various types of activation

Jack proposes various types of activations

1. Synchronous
2. Asynchronous
3. Free-wheel
4. Pipelined
Outline

1. Jack

2. Faust
   - Overview
   - Parallel code generation
   - Performances

3. Conclusion
Outline

1. Jack

2. Faust
   - Overview
   - Parallel code generation
   - Performances

3. Conclusion
FAUST: Functional AUdio Stream
A programming language for realtime signal processing

Design Principles:
Design Principles:

1. *Functional approach*: A purely functional programming language for real-time signal processing
Design Principles:

1. *Functional approach*: A purely functional programming language for real-time signal processing

2. *Strong formal basis*: A language with a well defined formal semantic
Design Principles:

1. *Functional approach*: A purely functional programming language for real-time signal processing

2. *Strong formal basis*: A language with a well defined formal semantic

3. *Efficient compiled code*: The generated C++ code should compete with hand-written code
Design Principles:

1. *Functional approach*: A purely functional programming language for real-time signal processing

2. *Strong formal basis*: A language with a well defined formal semantic

3. *Efficient compiled code*: The generated C++ code should compete with hand-written code

4. *Easy deployment*: Multiple native implementations from a single Faust program
A Faust program describes a *signal processor*, a mathematical function that maps input signals to output signals.

**Example**

```
process = +;
```
A Faust program describes a *signal processor*, a mathematical function that maps input signals to output signals.

Example

\[ \text{process} = +; \]
Stereo Pan

Faust syntax is based on a block diagram algebra:

\[(A:B), (A,B), (A<:B), (A:>B), (A\sim B)\]

```
Stereo Pan

\[p = hslider("pan", 0.5, 0, 1, 0.01);\]
\[process = _ <: *(sqrt(1 - p)), *(sqrt(p));\]
```
Stereo Pan

Faust syntax is based on a *block diagram algebra*: 

\[(A:B), (A,B), (A<:B), (A>:B), (A~B)\]

```plaintext
p = hslider("pan", 0.5, 0, 1, 0.01);
process = _ <: *(sqrt(1 - p)), *(sqrt(p));
```
Easy Deployment
Several audio platforms are supported

Thanks to specific *architecture files* the same Faust code can be used to generate a variety of applications or plugins:
Easy Deployment  
Several audio platforms are supported

Thanks to specific *architecture files* the same Faust code can be used to generate a variety of applications or plugins:

1. LADSPA
2. Max/MSP
3. Puredata
4. Q
5. SuperCollider
6. VST
7. Jack
8. Alsa
9. OSS
Some environments have Faust embedded
Some environments have Faust embedded

1. Snd-Rt: http://www.notam02.no/arkiv/doc/snd-rt/
   (see Kjetil Matheussen poster, August 27 - session 2)
Some environments have Faust embedded

1. Snd-Rt: http://www.notam02.no/arkiv/doc/snd-rt/ (see Kjetil Matheussen poster, August 27 - session 2)
2. CLAM: http://clam.iua.upf.edu/
Efficient code generation (monoprocessor)
Comparing Marteen de Boer’s Tapiir with the equivalent Faust Tapiir
Outline

1. Jack
2. Faust
   - Overview
   - Parallel code generation
   - Performances
3. Conclusion
two 1-pole filters in parallel connected to an adder

As an example we will use a very simple Faust program:

\[
\text{filter}(c) = *(1-c) : + \sim *(c); \\
\text{process} = \text{filter}(0.9), \text{filter}(0.8) : +;
\]
two 1-pole filters in parallel connected to an adder

Block-diagram representation automatically generated by Faust compiler using \texttt{-svg} option
The generated C++ code

The Faust compiler can produce 3 types of C++ code:

1. scalar code (default mode) see
2. vector code (-vec option) see
3. parallel code (-omp option) see
Outline

1. Jack

2. Faust
   - Overview
   - Parallel code generation
   - Performances

3. Conclusion
Multicore technologies in Jack and Faust

Faust
Performances

Freeverb
Freeverb on Vaio VGN-SZ3VP (2 cores)

Best speedup for the parallel version: 2, average: 1.62
Karplus 32

32 slightly detuned Karplus-strong strings mixed on a stereo bus.
Karplus 32 on Vaio VGN-SZ3VP (2 cores)

Best speedup for the parallel version: 1.59, average: 1.37
Wave Field Synthesis

Simple 8 channels Wave Field Synthesis.
Wave Field Synthesis on Vaio VGN-SZ3VP (2 cores)

Best speedup for the parallel version: 1.55, average: 1.17
Sonik Cube

Sound and Visual Installation (Trafik/GRAME, 2006) : 3mx3mx3m cube reacting to sounds in an audio feedback space
Toplevel block-diagram of Ethersonik, the audio software of Sonik Cube
Ethersonik

Voice block-diagram

```
voice(0) -> hgroup("voice 0")

oscillating_amp.dsp -> sonik_autocontrol.dsp -> autospat1x8.dsp
```
Ethersonik

Ethersonik source code

```plaintext
1  voice (v) = hgroup("voice %v",
2     component("oscillating_amp.dsp")
3     : component("sonik_autocontrol.dsp")
4     : component("autospat1x8.dsp")
5     );
6
7  M = vslider ("master",0,0,2,0.01);
8
9  process = hgroup("",
10     tgroup("", par(i,4,voice(i)))
11     => par(i,8,*M));
12```

Multicore technologies in Jack and Faust
Faust
Performances
Ethersonik on Vaio VGN-SZ3VP (2 cores)

Best speedup for the parallel version: 1.94, average: 1.79
Multicore technologies in Jack and Faust

Faust

Performances

Ethersonik on Macpro (8 cores)

Best speedup (cores: 5, vectors: 1024): 3,09, average: 2.88
Outline

1. Jack

2. Faust
   - Overview
   - Parallel code generation
   - Performances

3. Conclusion
To Sum Up

1. There is a lot of task + data parallelisms to exploit in audio applications.
2. Signal languages with a simple and well defined formal semantic are easy to parallelise. It's the way to go.
3. OpenMP is a simple and effective solution for multicore machines.
4. But efficient parallelisation is not that easy to achieve.
5. Memory bandwidth is a major limitation in SMP machines.
To Sum Up

1. There is a lot of task + data parallelisms to exploit in audio applications
To Sum Up

1. There is a lot of task + data parallelisms to exploit in audio applications.

2. Signal languages with a simple and well defined formal semantic are easy to parallelise. It’s the way to go.
To Sum Up

1. There is a lot of \textit{task + data} parallelisms to exploit in audio applications.
2. Signal languages with a \textit{simple and well defined formal semantic} are easy to parallelise. It’s the way to go.
3. OpenMP is a simple and effective solution for multicore machines.
To Sum Up

1. There is a lot of task + data parallellisms to exploit in audio applications
2. Signal languages with a simple and well defined formal semantic are easy to parallelise. It's the way to go.
3. OpenMP is a simple and effective solution for multicore machines
4. But efficient parallelisation is not that easy to achieve
To Sum Up

1. There is a lot of task + data parallelisms to exploit in audio applications.
2. Signal languages with a simple and well defined formal semantic are easy to parallelise. It’s the way to go.
3. OpenMP is a simple and effective solution for multicore machines.
4. But efficient parallelisation is not that easy to achieve.
5. Memory bandwith is a major limitation in SMP machine.
Ressources

1. Jack http://jackaudio.org
4. OpenMP http://openmp.org/wp/
5. Snd-Rt http://www.notam02.no/arkiv/doc/snd-rt/
6. CLAM http://clam.iua.upf.edu/