MultiConference Invited Talk

Multiprocessor Csound: Audio-Pro with Multiple DSP's and Dynamic Load Distribution

Barry Vercoe
Media Lab, MIT
Cambridge, MA
U.S.A.

Michael Haidar
Analog Devices Inc
Norwood, MA
U.S.A.

Hidehito Kitamura
Taito Corporation
Tokyo 1028648
Japan

Singaram Jayakumar
Epigon Audio Pvt.
Bangalore 560 008
India

Abstract - The latest professional Karaoke system released in Japan has no ASIC for sound synthesis and effects processing, but instead a small group of load-sharing DSP chips that cooperatively handle the varied and dynamically varying tasks of complex high-quality audio performance. The software-only system is a first for the professional audio industry, heralding a new generation of downloadable and task-sensitive software that delivers time-critical performance from distributed general-purpose silicon. The tasks of emulating a 64-voice orchestra plus real-time MPEG decode, live voice tracking with pitch and tempo following and a full range of audio effects processing are represented in a network of active objects which are just-in-time serviced by a cooperating array of SIMD DSP's. A detailed description of the system will conclude with a brief live performance.

1 Prelude

The domain of digital audio has lately become a battle-field of competing formats and representations (PCM, MP3, AAC, AC-3, MPEG-4), each of them putting a stake in the ground to claim the ideal balance between compression ratios (affecting transmission and storage) and computational complexity (compute power required at the client end). The fickle public, always ready to trade up to the next fad, wants to keep pace with the fast-moving content creator/providers who are eager to take advantage of the latest audio effects and to distribute their wares in whichever format seems to have its stake most solidly in the ground.

Audio hardware manufacturers are caught in the middle. They have a development period and time-to-market that lags behind the rest of the music industry. And because their solutions often take the form of application-specific devices (ASICs), they can find that pursuing quality and robustness can also flirt with obsolescence. The hardware industry must adopt technologies that are continually flexible and scalable, so that they can move with the outer ends of the industry that traffic in new content and new patterns of listening.

2 The Multiple Tasks of a Comprehensive Karaoke System

A compelling challenge is found in the Karaoke industry, where impending market saturation has caused a search for new functionalities within established tradition.

One missing functionality is in fact not new, but was an integral part of Karaoke tradition in its earlier form. The first Karaoke bars were Piano Bars, where a musician skilled at playing the standards popular at the time (such as Enka, similar to the show tunes of the American 40's) would provide sympathetic accompaniment for an amateur singer. This gave the singer an opportunity to bare his soul to his colleagues, to "ham it up" or "hang on a note or word", knowing that the pianist would lend full dramatic support. That functionality—carefully following the singer by constantly varying the tempo—disappeared entirely when technology got into the act, and the pianist was replaced by a machine.

A new functionality has arrived in the form of Background Chorus. This is a prerecorded audio clip which periodically joins the singer-soloist at various points in the song. For storage reasons the audio clip is commonly encoded in MP2 or MP3 (i.e. MPEG 1 layers 2 or 3). This means the Karaoke system must include a real-time MPEG decoder to reconstruct the steady stream of pre-recorded PCM audio. Switching the decoder on and off at the right time with the right file is accomplished by an extra MIDI track, another "voice" in the comprehensive MIDI file that drives the synthetic orchestra.

A few moments thought on what these two functionalities bring will reveal that they are basically incompatible. An MP3 file decoded to PCM audio has a fixed and predetermined musical tempo, while a system following a singer does not. And rate-changing a PCM stream will result

in unwanted pitch changes. The two cannot co-exist without additional heavy signal processing.

A fully comprehensive Karaoke system should be able to follow the singer's tempo, be a 64-voice synthesizer, a reverb and audio-post effects processor, a voice harmonizer, an equalizer (EQ), and a word-prompter for the song-text. It might need to be a melody prompter, a wrong-note corrector, or eventually change the singer's voice into that of a famous star. It might do a few of these in parallel well, or it might suddenly be asked to do all of them at once the best way it can manage.

This is no longer a task for fixed hardware ASICs. This requires flexible and downloadable functionality running on an architecture that is scalable and load-sensitive. This is the new reality for the audio industry.

3 The Csound Environment

Csound is a software audio processing system widely used by the computer music community [1,2]. It allows a user-defined set of *instruments* (signal-processing networks of oscillators, spectral filters, time envelope shapers, effects processors) to be invoked by items in a *score* (a time-ordered event-list representing note-on and note-off commands plus effects processing controls).

An *instrument* is defined as a template from which the *Csound* monitor can construct any number of *instantiations* (i.e may be invoked multiple times in parallel) and comprises a threaded list of signal processing modules, each with a unique *state space* for every single instantiation.

A *score* is a collection of time-critical requests which can emanate from many different directions—an ascii event list, a pre-existing MIDI file, a real-time MIDI stream (electronic keyboard, controller, or streaming Ethernet), or a real-time event-generating program—or from any combination or all of these directions at the same time.

Additionally, *Csound* can analyze and respond to audio from live microphone input, from a streaming file on disk, or from a streaming network source—or from any or all of these sources at the same time.

The nature of audio-processing in a *Csound* orchestra is defined by its *instruments*. Each *instrument* template can be a model of some audio-processing algorithm such as wave-table synthesis, additive synthesis, FM synthesis, linear prediction, phase vocoder, formant (FOF) synthesis, wave-shaping, physical modeling. Other instruments may perform analysis of live input such as pitch tracking [1,3,4], which can control the individual pitches of a vocal Harmonizer. Another may analyze the live input to perform tempo tracking [1,3,4], which in turn will influence the tempo of events performed in the current score. Yet others might add audio effects such as spatialization and reverb. All of these instruments and their imbedded algorithms can be invoked at the same time, and there can be any number of instantiations of each instrument at any particular moment.

The independence of dozens of simultaneously performing instruments is guaranteed by the unique *state space* that defines each *instantiation*. When a single note is turned off, the *state space* of the sounding instrument may be returned to the memory pool and thus become available for *instantiation* of some other instrument type. This amounts to *continuous garbage collection*. In an implementation with limited memory (such as a real-time hardware synthesizer) there are numerous algorithms that can be invoked for streamlining this process.

All processing within *Csound* is done in floating-point arithmetic, and the conversion to and from fixed-point audio is done as the streams enter and leave the *Csound* environment. *Csound* processes audio at CD rates (44.1 KHz) and proceeds through chronological time by block-processing the audio in *control period* blocks of 1 to 10 milliseconds. Since each musical note will last for many such periods, this means that each score event and the *instantiated* instrument assigned to perform it can together be viewed as a *continuous active object* whose momentary performance deadline is the end of the current *control period*. In a dense part of a symphony orchestra simulation there may be hundreds of these objects alive at any one moment.

The original *Csound* is now an Open Source standard used the world over [2]. A version of *Csound* called *NetSound* has become the core technology in MPEG-4 audio [5,6]. Some applications of *Csound's* unique *spectral data types* can be found in [3].

4 Extended Csound—Enabling a DSP as an All-Purpose Real-Time Audio Processor

Many tasks of *Csound* processing are not well-suited to general-purpose serial processors, and some of these—converting between time-domain and frequency-domain signals and invoking several iterations of small loops of code—will give an array processor with built-in butterfly hardware, SIMD processing power, and a non-trivial amount of on-chip memory a distinct advantage over serial processors. This is especially the case for a chip that specializes in high-speed floating point processing.

In 1995 *Csound* was ported to the *Analog Devices ADSP-21060*, for which ADI developed a series of hosting PCI boards that included audio codecs, and Uarts suitable for real-time MIDI I/O. The sudden propelling of Csound into *real-time interactive mode* engendered an explosion of its audio *Opcode* repertoire. Moreover, its ability to handle MIDI files and streams was greatly extended. This new version now with over 300 opcodes became known as *Extended Csound* [7].

Some advantages were immediate. The lightning response

of this new system was evident when it was played as a keyboard *synthesizer*. The best keyboards in the industry have a keystroke-to-sound delay of less than 5 milliseconds. The delay for *Extended Csound* is just two control periods—one for the active object processing described above, one-half for injecting the MIDI event into the active event list, and one-half for placing the resulting audio in the buffered output channel. When the control period is set to 1 msec, keyboard response is 2 msecs—a response unknown in the industry.

One innovation was especially effective here. The *MIDI Manager*, a program that fields incoming commands and resends them to the synthesizer units, was not relegated to a host microprocessor as in systems that incorporate ASIC's for their horsepower. Instead it is DSP-resident and interrupt driven, so that incoming events are immediately instantiated and inserted into the threaded list of active instruments. The *MIDI Manager* is basically a resource-sharing object amongst other instantiated objects.

Other advantages stem from the independent instantiation of all active objects (see above). Commercial synthesizers are typically built upon a single audio-processing method (DX-7 FM, Roland LA synthesis, Kurzweil and Ensoniq wave-table, Korg physical model), and though the computational complexity of a single voice is different for each synthesizer, it is the same for each note the instrument plays, so that the computational requirement for each note is trivial to predict.

Extended Csound is different. Since it can be *any* or *all* of the above algorithms *simultaneously*, each active note will put a different load on the resources depending on its specific computational complexity. To deal with this, each instrument *prototype* has built-in load-sensing code that can dynamically measure the computational cost of its instantiations. When the Csound monitor senses it is falling behind (i.e the output DAC buffer is emptying much faster than it is being filled), it can accurately and gently (i.e. silently) remove just the right number of inner overloading voices for stability to be regained.

The success of *Extended Csound–ADSP* combination in professional demonstrations soon led to requests that it be directed at tasks requiring considerably more compute power.

5 Multiprocessor Csound: Dynamic Load Distribution amongst Multiple Resources that can handle Unpredictable Demands

The *ADSP 21000* series of signal processors was designed to run co-operatively over a dedicated external bus that could accommodate up to six processors in tight synchrony. The on-chip memory of each processor has a unique bus address, and DSP-to-DSP communication of semaphores and DMA blocks of data can be in either single-target or broadcast mode. Also, any processor can reach inside the status registers of any other processor to find what is going on.

The applications that ADI had in mind were compute intensive, either from *breadth* of data to be processed or the *depth* of processing required. In either case a task is typically subset in advance, and the data sent by DMA from one processor to the next until the summary task is completed on schedule.

In *Csound* processing, the load is unpredictable and the network of dependencies is erratic. A keyboard player may put his whole forearm on the instrument, or a new MIDI file may suddenly request an entire change of voice models and audio effects, or a singer may suddenly slow down while simultaneously dialing a higher pitch transposition with full voice harmonizer effects.

A task-list of varying and unpredictable length is easily accommodated by the threaded list of *active objects* described above. The challenge is to direct the compute power of six DSP's onto this dynamically varying task load. The solution lies in organizing the DSP's in a Master-Slave relationship, modified to meet the inter-dependencies of certain tasks and the strict real-time deadlines of *control-period* processing.

At the start of each *control period*, the Master processor fields all incoming MIDI commands and score events, and updates the list of instruments potentially active. Any new instantiations are *initialized* at this time, meaning that their new *state memory* space is allocated, any sampling oscillators will be given compute cycles to locate their samples and reset their phases, new filter coefficients will be determined, and reverberators will allocate and clear their space to zero. All of this happens in *zero simulated time,* since no output samples are generated. And since there may be 20 or 40 new *Midi* and *score* events at any one moment, this initializing task is efficiently shared amongst all the processors available.

Once initialization is complete, the Master next divides the list of *active instrument* tasks between all available processors. This is done with pre-knowledge of the load being placed on each one since (as in the single-DSP version) each active object template maintains an estimate of the load it will incur, calculated during preceding control periods. This requires care, since some objects depend on others for data, and the most intensive tasks should ideally be distributed first. The Master will try to assign itself the least work, since it must also field interrupts from MIDI event arrivals and completed DMA transfers. The set of DSP's can now begin audio-generating performance.

If this dynamically balanced task list is found to place too much load on the multi-processor resources, causing the audio output buffer to drain, the same technique of *voice-stealing* for single-processor applications works equally well here. The Master processor will redistribute the currently active objects whenever the threaded task-list changes, and while this is normally due to notes being started and stopped by the MIDI stream and score file, a soft-ware stolen note will also trigger the same redistribution of resources.

A *Csound* active object list has a *tree* structure, in which

the large bulk of *generating* objects will gradually combine their outputs and forward them to a lower network of *effects* objects (delays, reverbs, etc). This incurs *dependencies* at some nodes of the tree, and this must be understood by a resource-conscious Master scheduler. Other dependencies will derive from that fact that shared input signals (voice mic and incoming audio streams) must be broadcast to all dependent objects. Ultimately the path is towards a stereo pair of effects-enhanced audio outputs, arriving on time before the close of the current *control period*.

The passage of musical time results from a *succession* of *control periods*, each receiving the resources it needs to complete its tasks on time. A few moments thought will reveal that every active *audio object* may be assigned to a different DSP on successive *control periods*, and this is true. In fact a single note of a melody may be successively generated by all six DSP's in just six control periods of a few milliseconds each, and yet the melody note must emerge clean and without blemish. This is possible due to the fact that *Multiprocessor Csound* is truly *object-oriented*, and the threaded task list is dynamically distributed between an arbitrary number (1 to 6) of parallel cooperating processors.

6 Time-Smearing the Task Depth

Some refinements to this structure should be mentioned at this point. First, merging several audio streams in a *Csound* tree requires that all contributing processors must first achieve *sync*. This is accomplished with the Csound *sync* command, which forces all denoted resources to *quiesce* before merging begins. Several levels of sync may be active at a single time, with several merges possible in parallel.

Second, shared use of the dedicated bus for interprocessor DMA's and access to external blocks of data require secure and fair arbitration. Since the tasks distributed to each processor at the start of a control period will also determine the order of DMA's in each, a threaded DMA list enables software chaining of DMA's in the order they are needed. Competing DSP's will then participate in a software token-passing scheme that assures that *bus-lock* is distributed efficiently and without competitive thrashing.

Although graceful degradation of a large MIDI score is handled by the *voice-stealing* described above, being able to keep all DSP's maximally busy is a preferred strategy. A typical *Csound* tree may have a hundred simultaneous objects near its top but only one or two large ones (reverbs) near its base, which can leave some processors idle while a smaller number work to complete the tree. Using tree-relevant directives in the Orchestra template, compute-intensive *effects objects* can be folded back to run at the top of the next *control period*. While this complicates the tree, it enables *maximum utilization of resources*, and its realization fits neatly into the structures of *Dynamic Load Distribution* on which *Multiprocessor Csound* is based.

7 A Major First for the Audio Industry

The impetus for developing an efficient *Multiprocessor Csound* came from two Japanese companies, Denon (who first licensed the technology from Analog Devices) and Taito Corporation (the fourth largest Karaoke manufacturer in Japan and the first to introduce Communication Karaoke). Multiprocessor board design and software support was provided by Epigonaudio. The goal of this initiative was to bring the flexibility of software-only audio processing into an otherwise rigid ASIC-dominated industry.

The result is a new Taito system called the *Lavca*. It uses 3 ADSP 21161 SIMD processors amassing 1.8 Gigaflops (peak) of tightly-coupled multiprocessing to deliver professional Karaoke performance including 64-voice MIDI synthesis, on-the-fly MPEG decodes, full effects processing, pitch and tempo modification, dynamic EQ, voice tracking and enhancement such as following tempo changes and correcting wrong pitches. The audio system is paired with a custom video system, and both are supported by an audio subsystem, various video monitoring and display devices, and a host interface that can access data streams via internet and satellite.

The absence of audio-processing ASIC's in a professional audio product is a *first* for the audio industry. Although the system is available only in Japan, this paper will conclude with a live demonstration of the flexibility and power of its comprehensive software-only audio-processing.

8 References

[1] Vercoe, B.L., and Ellis, D.P.W. (1990) "Real-time Csound: Software Synthesis with Sensing and Control," in *Proceedings, ICMC*, Glasgow, pp. 209-211.

[2] Boulanger, R.C. (Ed.), *The Csound Book*, Cambridge, MA: MIT Press, 2000.

[3] Vercoe, B.L. (2000) "Understanding Csound's Spectral Data Types," in Boulanger, R.C. (Ed.), *The Csound Book*, Cambridge, MA: MIT Press, pp. 437-447

[4] Vercoe, B.L. (1997) "Computational Auditory Pathways to Music Understanding," in Deliege I. and Sloboda J. (Eds.), *Perception and Cognition of Music*, East Sussex, UK: Psychology Press, pp. 307-326.

[5] Vercoe, B.L., Gardner, W.G., and Scheirer, E.D. (1998) "Structured Audio: Creation, Transmission, and Rendering of Parametric Sound Representations ," in *Proceedings of the IEEE* 86:5 (May 1998), pp. 922-940.

[6] Scheirer, E. D. and Vercoe, B.L. (1999). "SAOL: The MPEG-4 Structured Audio Orchestra Language," *Computer Music Journal* 23:2 (Summer 1999), pp 31-51.

[7] Vercoe, B.L. (1996) "Extended Csound," in *Proceedings, ICMC*, Hong Kong, pp 141-142.