Audio-pro with multiple DSPs and dynamic load distribution

B Vercoe

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 DSPs that co-operatively handle the varied and dynamically varying tasks of complex high-quality audio performance. The software-only system is a first for the 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 co-operating array of SIMD DSPs.

1. Introduction

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 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 1940s) 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, 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 controlled by a special 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 into PCM audio has a predetermined musical tempo, whereas a system following a singer does not, and rate-changing a PCM stream will result in unwanted pitch changes. The two functionalities 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 synthesiser, a reverb and audio-post-effects processor, a voice harmoniser, an equaliser (EQ), and a word-prompter for the song-text. It might need to be a melody prompter, a wrong-note corrector, or 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 in 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. 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.

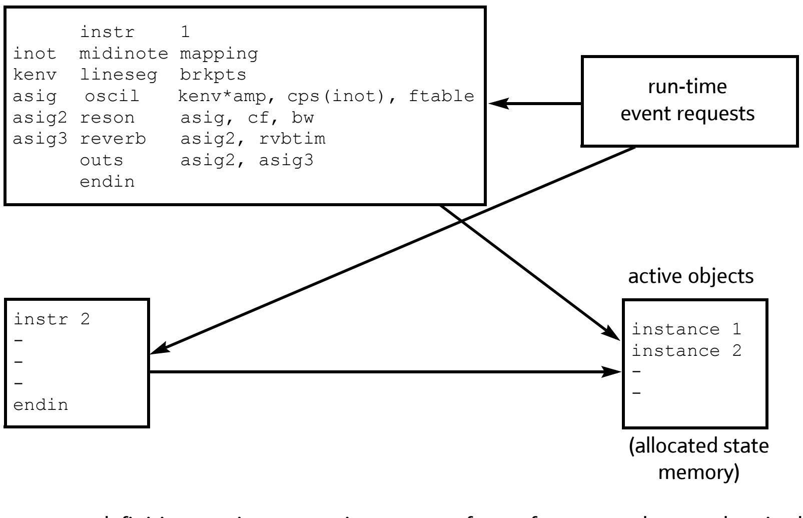
Additionally, Csound can accept and respond to audio from live microphone input, from a streaming file on disk, or from a streaming network source, or from any or all of these sources at the same time.

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 modelling. Other instruments may perform analysis of live input such as pitch tracking [1—3], which can control the individual pitches of a vocal Harmoniser. Another may analyse the live input to perform tempo tracking [1—3], which in turn will influence the tempo of events performed in the current score. Yet others might add audio effects such as spatialisation 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.

A simple Csound instrument and its activation are depicted in Fig 1. In the instrument definition the terms midinote, lineseg, oscil, reson, reverb and outs are opcodes, each of which operates on the input arguments to its right and places an output signal in the cell to its left. Output signals can then become inputs of other subsequent opcodes. For ease of reading, this instrument contains the mnemonics mapping, brkpts, amp, ftable, cf, bw, rvbtim in place of less meaningful real input values, but the opcode linkages are clear. At run-time, this instrument definition is only a template without any real memory space. It becomes instantiated when the Music Monitor receives a Note Event Request (from a score or MIDI device), at which time the instrument is allocated a block of state memory space in which its opcodes can perform. This block is then inserted into a threaded list of active objects, to be performed until the note event is eventually turned off.

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 is equivalent to continuous garbage collection. In an implementation with limited memory (such as a real-time hardware synthesiser) 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 normally processes audio at CD rates (44.1 KHz) and proceeds through chronological time by block-processing the audio in control period blocks of usually 1 to 10 msec. 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 deadline is the end of the



```
      instr    1
inot  midinote mapping
kenv  lineseg  brkpts
asig   oscil   kenv*amp, cps(inot), ftable
asig2 reson    asig, cf, bw
asig3 reverb   asig2, rvbtim
      outs     asig2, asig3
      endin
```

```
instr 2
-
-
-
endin
```

```
instance 1
instance 2
-
-
```

Fig 1 When a Csound instrument definition receives a run-time request for performance, the template is allocated state memory that enables its opcodes to perform, and the memory block is inserted into a threaded list of active objects to be run until the note is eventually turned off.

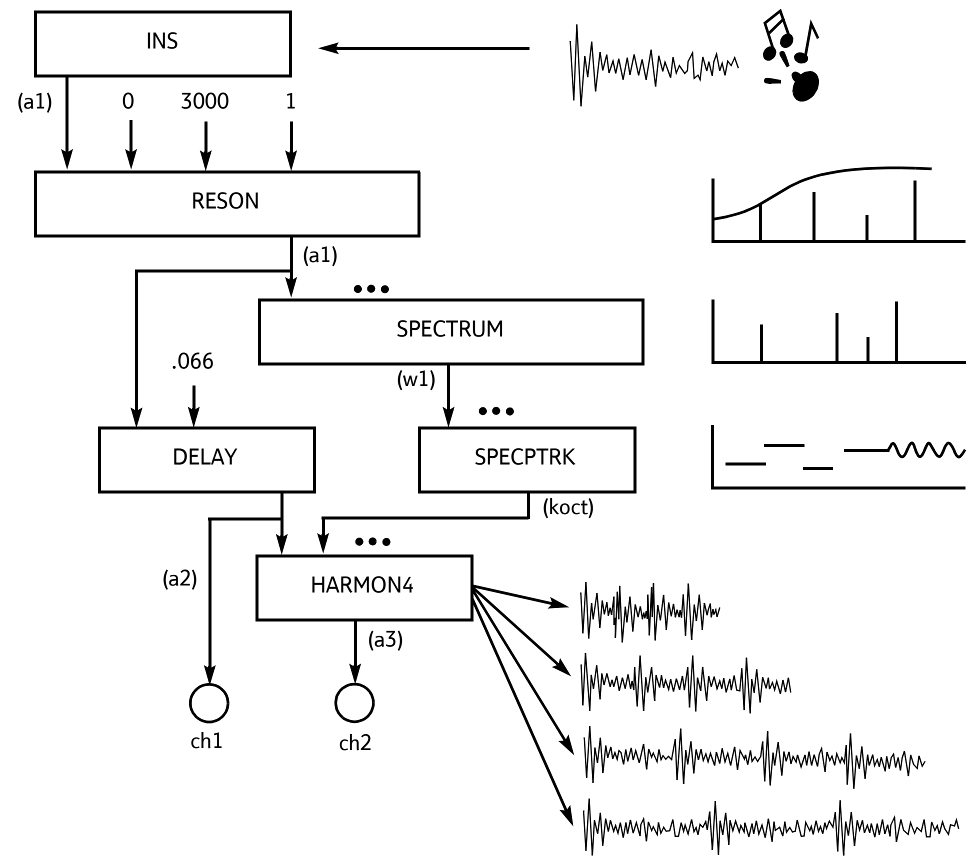
current control period. In a dense part of a symphony orchestra simulation there may be hundreds of these objects alive at any one moment.

4. Spectral data types

While most opcodes in Csound are for signal generation, to promote constants and control-rate signals refreshed a few hundred times per second into audio-rate signals refreshed more than forty thousand times per second, the purpose of spectral data types is quite the opposite — to enable incoming audio signals to be analysed for their slower-moving control content. The data types are self-defining, containing in their structure the details of how often they have sampled the incoming source signal, and with what kind of frequency resolution.

Spectral information is not derived from a fast Fourier transform which has linearly spaced bins across its frequency range, but from exponentially spaced Fourier matchings in which a windowed segment of the incoming signal is multiplied by sinusoids of exponentially spaced frequencies. The resulting data is much closer to the information captured by the cochlear of the human auditory system, and the spectral data type thus more closely represents the information a human listener would perceive given the same source audio signal. Spectral data types are therefore ideal sources of information with which to do audio pitch tracking and musical rhythm detection.

A more complex Csound instrument that can pitch-track an incoming audio signal and turn that into a five-part harmony is shown in Fig 2. This first takes one channel of audio input and gives it some simple equalisation (EQ) to heighten the voice partials. The spectrum opcode then derives a spectral data type *w*1, collected every 0.02 sec, that has 24 frequency bins per octave and a bandwidth Q of 12 on each bin. The opcode



```
            instr     9                            ;PITCH TRACKING HARMONIZER
a1, a0      ins                                    ;GET MICROPHONE INPUT
a1          reson     a1, 0, 3000, 1               ;AND APPLY SOME EQ
w1          spectrum  a1, .02, 6, 24, 12, 1, 3     ;FORM A SPECTRAL DATA TYPE
                                                   ;FIND THE PIRCH
koct, kamp  spectrk   w1, 1, 6.5, 8.9, 7.5, 10, 7, .7, 0, 3, 1, .1
a2          delay     a1, .066                     ;TIME ALIGN PITCH & AUDIO
                                                   ;ADD 4 NEW PARTS
a3          harmon4   a2, koct, 1.25, .75, 1.5, 1.875, 0, 6.5
            outs      a2, a3                       ;AND SEND ALL 5 TO OUTPUT
            endin
```

Fig 2 A Csound pitch tracker and harmoniser. Input signal *a*1 is pre-emphasised, then analysed to produce an exponentially spaced frequency spectrum *w*1, which is ideal input for the pitch tracker spectrk. The pitch koct is then used by harmoniser harmon4 to create four additional voices at specified pitch intervals, all with the same vowel quality as the original.

is also requested to use a Hanning window on the input, and to emit root magnitude spectral data.

This can now send specptrk some favourable data, and its arguments request that it analyse the signal *w*1 across the octave range 6.5 to 8.9 (F below the piano bass staff to A above middle C) using an internal template of 7 harmonic partials with a roll-off of 0.7 per octave. They also request 3 confirmations of any sudden octave leap, and ask that the pitch and amplitude outputs 'koct' and 'kamp' be control-rate interpolated between consecutive analyses. Finally it is asked to display the running cross-correlation spectrum so that we can observe its various internal pitch candidates in dynamic competition.

The harmon4 unit is similar to the harmoniser in normal Csound, but borrowed here from Extended Csound [4] (see section 5 below) because of its additional features. Using the koct pitch information it will take *a*2 (a time-synchronised version of the original audio) and pitch-shift it four ways into a four-part chord of frequency ratios 1.25, 0.75, 1.5 and 1.875 (i.e. a dominant seventh chord), making 5-part harmony, with all voices preserving the vocal formants and vowel quality of the original audio input.

Some additional applications of Csound's unique spectral data types can be found in Vercoe [2, 3]. The original Csound is now an Open Source standard used the world over [5]. A version of Csound called NetSound has become the core technology in MPEG-4 audio [6, 7].

5. Extended Csound-enabling a DSP as an all-purpose real-time audio processor

Many aspects 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 specialises in high-speed floating point processing.

In 1995 Csound was ported to the analogue 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 [4].

Some advantages were immediate. The lightning response of this new system was evident when it was played as a keyboard synthesiser. The best keyboards in the industry have a keystroke-to-sound delay of less than 5 msec. 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 msec — a response unknown elsewhere in the industry.

One innovation was especially effective here. The MIDI Manager, a program that fields incoming commands and resends them to the synthesiser units, was not relegated to a host microprocessor as in systems that incorporate ASICs 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 can thus be viewed as another object, sharing the resources alongside other instantiated objects.

Other advantages stem from the independent instantiation of all active objects (see above). Commercial synthesisers 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 synthesiser, it is the same for each note the instrument plays, and the maximum number of simultaneous notes has a hard limit defined by the number of 'voice slots' in the hardware.

Csound is different. The maximum number of simultaneous notes has a soft limit depending on their complexity. While a software synthesis processor may reach its capacity in performing 20 or 30 simultaneous notes of great complexity, the same processor might easily perform 120 or 150 simultaneous notes of lesser complexity. In practice, a typical orchestra will be made up of a variety of instrument algorithms of different complexity. And since the score might call on any or all of these algorithms simultaneously, with each of them to some arbitrary multiplicity depending on the number of notes of that kind requested in parallel, the processor's theoretic capacity will change as often as there are calls for a new kind of note.

Extended Csound has a special way of dealing with this. Each instrument prototype contains built-in load-sending code that can dynamically measure the computational cost of each of its instantiations. When the Csound monitor senses the system is falling behind (i.e. the output DAC buffer is emptying much faster than it is being filled) it can accurately and gently remove (i.e. envelope out) just the right number of inner overloading voices for stability to be regained. This is an adaptive soft-limit version of the harsher voice-stealing property of fixed-slot ASIC synthesisers.

Following considerable extensions and development, Extended Csound was shown in numerous live performance demonstrations, including at the 1996 International Computer Music Conference in Hong Kong [4], and the 1996 AES Convention in Los Angeles where it was awarded the EQ Blue Ribbon Award for the Best Product at that Convention.

However, the success of Extended Csound-ADSP combination in professional demonstrations soon led to requests that it be directed at tasks requiring even more compute power.

6. Multiprocessor Csound — dynamic load distribution among multiple resources that can handle unpredictable demands

The above successes were followed by a special challenge in 1998 when Japan's Taito Corporation sought the flexibility of Extended Csound in a high-end professional audio product.

The goal was 64-voice MIDI synthesis, real-time tempo-varying MPEG decode, tempo-following voice tracking, comprehensive audio-post effects processing (chorusing, flanging, and numerous prescribed reverb configurations in 2 or 4 channels), in a software-only system that ran at 48 Khz. This required a small bank of tightly coupled DSPs, and a novel Multiprocessor Csound that could dynamically share the time-varying load across all available resources (see the realisation in Fig 3).

The ADI ADSP 21000 series of signal processors had been 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 standard applications that ADI had in mind for their ADSP 21000 were compute intensive, either from breadth of data to be processed or the depth of processing required. In either case the task would typically be 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 also dialling a higher pitch transposition with full voice harmoniser effects.

The resulting 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 DSPs on to this dynamically varying task load. The solution lies in organising the DSPs in a Master-Slave relationship, modified to meet the interdependencies 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 initialised at this time, meaning that their new state-memory space will be allocated, any sampling oscillators will locate their samples and reset their phases, new filter coefficients will be determined, and reverberators will allocate and clear their internal space to zero. All of this happens in zero simulated time, since no output samples are being generated. And since there may be 20 or 40 new MIDI and score events at any one moment, this initialising task is efficiently shared (for every note-onset) among all the available processors.

Once initialisation is complete, the Master next divides the list of active instruments between all available processors as depicted in Fig 4. 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 the 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 DSPs can now begin its audio-generating performance.

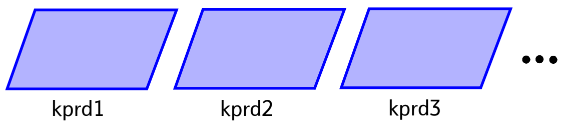


Fig 4 Overlapped processing of control-period audio tasks. While the tail of one kprd of audio processing (typically audio-post effects) is being completed, the head of the next kprd of audio is being assigned and worked on by the other processors. This guarantees maximum utilisation of processing resources.

If the dynamically balanced task list is found to place too much load on the multi-processor resources, causing the audio output buffer to drain faster than it is being replenished, the same technique used in voice-stealing for single-processor applications works equally well here. While the overloading voices are gradually being retired, the Master processor will continue to redistribute the currently active objects whenever the threaded task-list changes. Although this action is normally to handle notes being started and stopped by the MIDI stream and score file, a retiring note caused by voice-stealing will also trigger a similar redistribution of resources.

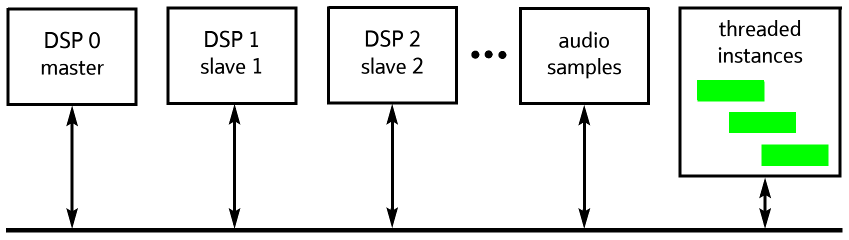


Fig 3 In Multiprocessor Csound up to six processors operate in a master/slave relationship to collectively service the needs of the active objects that represent the instantiated and currently active instruments. Each instrument template maintains a running estimate of the computational load that a single instantiation puts on a processor, so that the load of all instances of all instruments can be automatically redistributed each time a new note begins or an old note is removed from the threaded list.

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 the resource-conscious Master scheduler. Other dependencies will derive from the fact that a shared source of 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 in fact true. Moreover, a single note of a melody may be successively generated by all six DSPs in just six control periods of a few milliseconds each, and yet the melody note must emerge clean and without blemish. This is achieved because Multiprocessor Csound is truly object-oriented, and the threaded tasklist is dynamically distributed between an arbitrary number (from 1 to 6) of parallel co operating processors.

7. Time-smearing the task depth

Some refinements to this structure should be mentioned at this point. Firstly, merging several audio streams in a Csound tree requires that all contributing processors must first achieve sync. This is accomplished with the Csound sync opcode, 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.

Secondly, shared use of a dedicated bus for interprocessor DMAs and for access to external blocks of data requires secure and fair arbitration. Since the tasks distributed to each processor at the start of a control period will also determine the order of DMAs in each, a threaded DMA list enables software chaining of DMAs in the order they are needed. Competing DSPs 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 guaranteed by the voice-stealing described above, being able to keep all DSPs maximally busy is a concomitant goal. A typical Csound tree may have several 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.

This overlapped processing of control-period audio slices is depicted in Fig 5. While this complicates the tree, it enables maximum utilisation of resources, and its realisation fits neatly into the structures of dynamic load distribution on which Multiprocessor Csound is based.

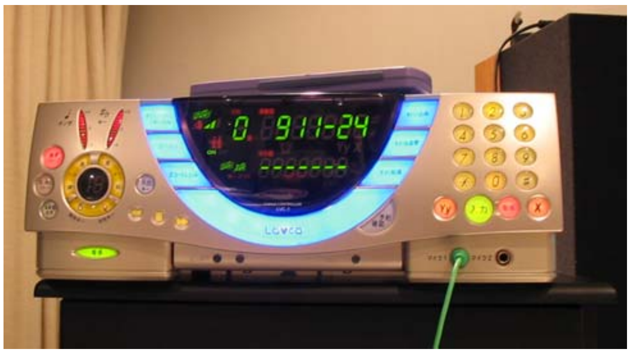


Fig 5 Taito Corporation's Lavca System, the audio industry's first professional audio product based entirely on software audio processing. Using Multiprocessor Csound running on 3 SIMD dual-core floating-point DSPs from Analog Devices, the system delivers 64 voices of MIDI synthesis, real-time tempo-varying MPEG decode, tempo-following voice-tracking, and comprehensive audio-post-effects processing, all at 48 kHz in 2- or 4-channel sound. Each system contains 40 000 songs, outputs high-quality graphics and synchronised text, and has a unique IP address for updating in the field.

8. 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 Epigon Media Technologies of Bangalore, India. The goal of this initiative was to bring the flexibility of software-only audio processing into an otherwise rigid ASIC-dominated industry.

The first result is a new Taito system called the Lavca, pictured in Fig 5. 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 audio-post effects processing, pitch and tempo modification, dynamic EQ, voice tracking and enhancements, 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 the Internet and over satellite. There are now 25 000 of these high-end commercial Lavca systems installed in the field. Each unit has an IP address, and the internal audio and video software is routinely updated by downloading over the net [8].

The absence of audio-processing ASICs in a professional audio product is a first for the audio industry. Although the Lavca system is currently available only in Japan, its immediate success suggests that the idea will soon spread elsewhere. This practical use of multiprocessor audio with dynamic load distribution shows that the flexibility and power of comprehensive software-only audio processing will play a significant role in the compute-intensive digital audio industry of the future.

References

1 Vercoe B L and Ellis D P W: ‘Real-time Csound: software synthesis with sensing and control’, in Proceedings, ICMC, Glasgow, pp 209—211 (1990).

2 Vercoe B L: ‘Computational Auditory Pathways to Music Understanding’, in Deliège I and Slobada J (Eds): ‘Perception and cognition of music’, East Sussex, UK, Psychology Press, pp 307—326 (1997).

3 Vercoe B L: ‘Understanding Csound’s spectral data types’, in Boulanger R C (Ed): ‘The Csound book’, Cambridge, MA. The MIT Press, pp 437—447 (2000).

4 Vercoe B L: ‘Extended Csound,’ in Proceedings, ICMC, Hong Kong, pp 141—142 (1996).

5 Boulanger R C (Ed): ‘The Csound book’, Cambridge, MA, The MIT Press (2000).

6 Vercoe B L, Gardner W G and Scheirer E D: ‘Structured audio: creation, transmission, and rendering of parametric sound representations’, in Proceedings of the IEEE, 86, No 5, pp 922—940 (May 1998).

7 Scheirer E D and Vercoe B L: ‘SAOL: the MPEG-4 structured audio orchestra language’, Computer Music Journal, 23, No 2, pp 31—51 (Summer 1999).

8 Vercoe B L, Haidar M, Kitamura H and Jayakumar S: ‘Multiprocessor Csound: audio-pro with multiple DSPs and dynamic load distribution’, in Proceedings of the Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas (2003).

[redacted]

B Vercoe