RTS: REAL TIME SCHEDULING IN COMMON MUSIC

Heinrich Taube
University of Illinois
School of Music
Urbana, Illinois

Todd Ingalls
Arizona State University
Arts, Media and Engineering
Tempe, Arizona

ABSTRACT

RTS is a real time scheduler for the Common Music algorithmic composition system. RTS enables compositional algorithms written in Lisp to generate sound in real time and to communicate with concurrently executing applications via OSC [1], Portmidi [2] and Midishare [3] connections. One of the nicest features of the RTS scheduler is that it can be "invisible" to musical algorithms, that is, a given musical algorithm is able to execute in either real time or non-real time modes without any change to its code. This allows a composer to use the identical suite of algorithms for both file-based and interactive work simply by switching scheduling modes, without having to change how the algorithms themselves are expressed. Of course real-time execution can limit how much computation can happen in a given execution slice and so the RTS system also provides a library of compositional tools specifically designed with real time execution in mind for composers that want to optimize their algorithms for real time use. The RTS scheduler is based on POSIX threads [4] and is implemented as an optional software system that can be loaded into Common Music on demand, whenever the composer wants to work in real time. RTS can run in any Common Lisp or Scheme that supports *callbacks* and foreign function calling. The current bindings support OpenMCL [5], SBCL [6] and Gauche Scheme [7] on both Linux and OS X. A version may be available for SBCL/Windows in the future.

1. INTRODUCTION

Common Music (CM) is an algorithmic composition system implemented in Common Lisp and Scheme. In Common Music, compositional algorithms are defined and then *sprouted*, or executed, to render musical results. These results can be directed to files, ports or displays by opening up the appropriate *event stream* to route the compositional data to the desired destination. Until RTS the execution of algorithmic processes happened in non-real time (typically many times faster than real time), which meant that Common Music could not be used in interactive performance environments, nor run cooperatively in conjunction with other (concurrently executing) external programs. This inability to support real time execution stemmed from the fact that the Lisp language specification does not address threaded execution or, more generally, language connectivity, i.e. the ability of a Lisp program to communicate (to call or be called) by a program written in a different language [8]. However, in the last several years a number of parallel efforts by Lisp developers have resulted in the ability for almost all Lisp implementations to connect to external languages using a foreign function interface. Chief among these efforts is the CFFI project [9], which defines a standard API and a stable foreign function code base that works with the vast majority of Common Lisp implementations available today. Common Music's RTS module leverages this work by defining a C based threaded scheduler that can be called by -- and can call back *into* -- a host lisp. Using this facility is now possible to compose algorithmically in real-time in several public domain Common Lisp and Scheme implementations available on Linux and OSX.

2. THE SCHEDULER

The RTS scheduler consists of Lisp and C code that is compiled and loaded into an executing Lisp environment. The C code is stored as a shared library produced by the C compiler. The Lisp side consists of a foreign function interface that defines lisp entry points in the C library, a registered callback that allows the POSIX scheduling thread to invoke Lisp instructions, and support code that maintains a registry of CM algorithms and sequences (vectors of event data) that are currently running in real time.

2.1. Starting, Stopping and Querying the Scheduler

Once the RTS system has been loaded into Common Music the RTS scheduler is in one of three states: stopped, running or paused. To start the scheduler, the Lisp function (rts) is called. This function can be passed several optional pieces of information: (1) a *time format*, either floating point seconds or integer milliseconds; a *thread priority* (1-100) that determines the relative execution priority compared to other OS threads; a default *output stream*, or destination for the musical events that are generated under RTS, a scheduling resolution that controls how often the thread will examine the queue (defaults to .1 millisecond), and a POSIX scheduling *policy* that defaults to Round Robin. A running RTS scheduler can be paused and continued interactively using the (rts-pause) and (rts-continue) functions, and stopped altogether using (rts-stop). The function (rts?) can be used to query the scheduler as to its current execution state and the function (now) will return the current clock time of the scheduler as either floating point seconds or integer

milliseconds, as specified when the scheduler was started.

2.2. Sprouting

Once the scheduler has been started, any Lisp object (CLOS objects, function calls, musical algorithms, sequences of event data, etc) can be added to the scheduler interactively, by calling the (sprout) function to insert the object into the scheduler. The sprout function also supports an optional *ahead* factor that will cause the sprouted object to be inserted into the future from the time of the sprout. When a Lisp object is spouted it is added to a hash table under a unique integer *key* that will allow the object to be very quickly recovered via integer hash lookup when its time is due. The handle is then passed over to the C side and entered in to the scheduling queue.

2.3. Real time Execution

A Lisp algorithm executing in real time can output data to an open output stream, query the current scheduling time to make decisions, sprout new objects (algorithms and sequences) to the scheduler and suspend itself for a future wakeup time.

2.4. The C Scheduler

The C scheduler is implemented as a POSIX thread that runs in parallel with Lisp. When an object is sprouted on the Lisp side the sprout function locks the queue and calls a C function to insert event data into the scheduler. The scheduling queue is a time sorted, linked list of queue nodes. The scheduler maintains a pointer to the last node in the queue as well as the first to optimize for the very common case of appending (later) time events to the end of the queue as well as retrieving events at the beginning of the queue.

2.5. Queue Nodes

Each queue node consists of a microsecond time stamp, a data field and a pointer (index) to the next node in the queue. The node's data field can contain *immediate* data (data that can be sent directly to an output stream at the appropriate time) or a *handle* to a dynamic Lisp object to be executed back inside Lisp when its time stamp becomes current. A handle is simply an integer encoded with type information that uniquely identifies the Lisp object to be processed. This object can be a musical process, a Lisp function or a sequence (container) of Lisp CLOS objects that define parameterized musical event descriptions.

Queue nodes are maintained in a statically allocated table (a compile time flag allows the number of queues to be increased or decreased when the scheduler is installed). The first node in the table is reserved as a global pointer to the next available (free) node the scheduler can allocate or to NULL if there are no more free nodes or the queue is empty. When the scheduler is started it marks all nodes in its table as free and links them together to form a *free node list*. As data is sent to the scheduler the next free node is accessed in constant time, its free pointer becomes the first node's next free node, and the newly allocated node is assigned the timestamp and data passed into the scheduler from Lisp. This newly allocated node is then inserted into the queue at the latest possible time according to its time stamp; nodes added later with the same time stamp appear later in the queue list.

2.6. The Scheduling Thread

The RTS scheduling thread runs concurrently with the main Lisp thread; if there are no queue nodes the thread blocks until something has been inserted by a sprout operation. It then starts examining the queue at the rate of the scheduling resolution (by default ten times per millisecond) in order to pop nodes off when their time stamps become current. If the current (popped) node contains immediate data, that data is sent to the open output stream for sound rendering. If the node is a handle to a dynamic lisp object such as a musical process or a function call the handle is sent back to Lisp at the correct scheduling time via the *callback* mechanism described in the next section. Once the popped node has been processed the thread sleeps for the timing resolution and then examines the queue again.

2.7. The Lisp Callback

A lisp callback is a registered entry point into the Lisp environment that can be invoked by foreign functions to asynchronously evaluate Lisp expressions. Not all Lisps support callbacks, but most of them do, including three high-quality, public domain implementations (SBCL, OpenMCL and Gauche Scheme). The Lisp callback that RTS establishes enables the POSIX scheduling thread to send the current thread time and the handle to a Lisp object kept in the queue node to be sent back to Lisp asynchronously for processing. The callback processes this data by first accessing the associated Lisp object using the integer handle as the hash key for a hashtable lookup, and then evaluating the associated Lisp object according to its type byte encoded in the handle. For example, function objects and musical algorithms (lexical closures) are immediately *funcalled* (a Lisp term for applying a function or lexical closure to zero or more arguments) in order to trigger whatever action the composer designed them to perform. Since this always involves executing user code, the RTS callback wraps this funcall inside an error handler that catches any Lisp error triggered under the callback. The RTS error handler will print the Lisp error condition and then allow the user to either (1) drop the offending entry from further scheduling or (2) stop the RTS scheduler altogether. If the Lisp object is a musical algorithm, funcalling will likely result in more objects being sprouted or rescheduled back into the C queue. In this way the scheduler processes data back and forth between Lisp and the POSIX thread as long as the queue is not empty or the scheduler is not paused or stopped.

3. REAL TIME OUTPUT STREAMS

Common Music provides several different types of event streams for working in real time. These streams

provide a "direct connection" to external device drivers or concurrently executing software application. Real time streams can be loaded "on demand" into CM to establish these external connections via a CFFI foreign function interface. Common Music's Portmidi and Midishare streams provide MIDI input/output services and support the complete APIs of their target software systems. The OSC stream implements sending/receiving OSC messages to a UDP socket using the Open Sound Control protocol. This allows Lisp algorithms running in real time to control (or be controlled by) synthesis patches running in external applications such as SuperCollider or Pd (Figure 2).

4. INPUT HOOKS

In addition to the real time output capabilities of RTS, a simple API is defined for responding to events from real time input streams such as Portmidi, Midishare or OSC. Each class of input stream has a dynamic library defined for it which can launch a *listening thread*, either in blocking or non-blocking mode depending on the underlying model of the input protocol. This thread can be started by the (recv) function. The priority of this thread can be set as well as its polling resolution, in the case of non-blocking input methods (for instance Portmidi), when the thread is started.

Once started a user definable function or *hook* can be registered for this input stream with the (recv-set!) function (see Figure 2). This function or closure will be funcalled each time a new event is available in the input stream. The current status of the input thread can be queried with the (recv?) function and stopped with (recv-stop).

Conceptually this same mechanism could be applied to almost any type of input, including input received from GUI elements, changes in other musical algorithms or even through more complex analysis of a combination of inputs.

5. GARBAGE COLLECTION

Garbage collection (GC) refers to a type of automatic memory management that periodically reclaims allocated but unused memory. Every Lisp implementation must provide garbage collection, but the Language specification does not specify what type(s) of GC the implementation must support. Since GC can have an impact on musical processes running in real time this is an important consideration when choosing which Lisp to use with RTS. Some Lisp implementations such as OpenMCL provide a very fast, non-invasive GC called *ephemeral* GC (also known as generational GC) which are based on the idea it is more efficient to focus memory reclamation efforts on the most newly created objects [5]. Others use sophisticated locking and synchronization mechanisms to GC in parallel with the main application thread, such as the Boehm-Demers-Weiser GC [13] used by the Linux version of Gauche scheme. Still other Lisp implementations only provide a slower, stop-and-copy approach, such as Gauche scheme running under Mac OSX. In general, the more garbage a program generates the more a GC can be triggered, so real time programs should make efforts to avoid creating temporary, short lived data structures. Garbage can be eliminated fairly easily by pursuing strategies such as (re)using preallocated CLOS objects, using vectors rather than temporary lists and switching to integers for math calculation. The RTS system itself performs no runtime memory allocation and so is neutral with respect to garbage collection. It also provides a number of "consless" composition operators for performing common tasks such as random data generation, linear interpolation and so forth that do not use or trigger dynamic memory allocation.

Timing tests performed have demonstrated that even without too much care taken, timing and event scheduling can be run quite accurately as long as the Lisp being used has an efficient GC policy (Figure 1).

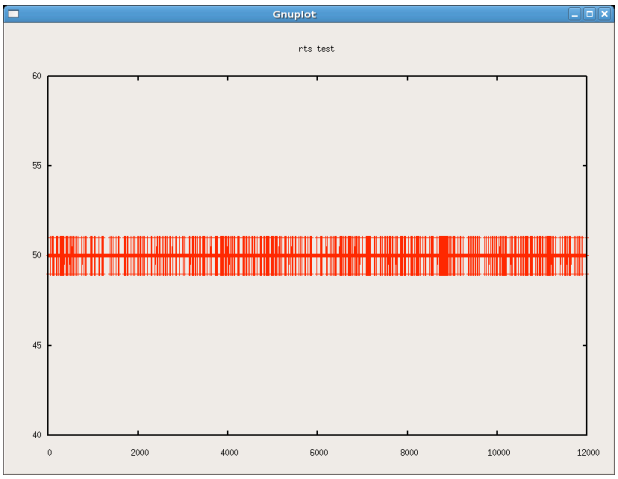

Figure 1. Timing results from scheduling a musical algorithm every 50 ms for 12000 iterations (10 minutes). Max deviation +-1 ms. Example run in SBCL 1.0 on Fedora Linux, Planet CCRMA distribution.

.

6. REAL TIME EXAMPLE

The following example program demonsrates how an algorithmic program is defined and executed in real time in Common Music.

```
(use-system :portmidi)
(use-system :rts)

(define fluff '(60 62 64 67 72 65 69 48 50))

(define *pm* (portmidi-open :input 0 :output
3))
```

```
(define (endless-fluff num dur knums)
  (process repeat num for i from 0
     output (new midi :time (now)
               :duration (* 2 dur)
               :amplitude .5
               :keynum (pickl fluff))
     wait (pick dur (/ dur 2) (/ dur 4))
     when (= i (1- num))
     sprout
     (process repeat 4
       output (new midi :time (now)
                  :duration 5
                  :amplitude .5
                  :keynum (pickl knums)))
     and
      sprout (endless-fluff 20 1 knums)))

(rts *pm*)

(sprout (endless-fluff 20 1 fluff))

;;; register and input hook to respond
;;; to incoming midi messages. This will
;;; sprout endless-fluff with num set to
;;; the incoming midi key number

(recv *pm*)

(recv-set! *pm*
     (lambda (midi-message)
       (if (note-on-p midi-message)
          (sprout
              (endless-fluff  (note-on-key  1
fluff))))))

;;; stop when you are done.

(recv-stop *pm*)

(rts-stop)
```

Figure 2. A recursive algorithm that can only run in real time: It outputs midi notes and then sprouts another copy of itself, providing continuous output. Also demonstrated is the use of an input receiving hook to sprout additional copies.

7. REFERENCES

[1] Wright, M., A. Freed, and A. Momeni. "OpenSound Control: State of the Art 2003." *Proceedings of the 2003 International Conference on New Interfaces for Musical Expression (NIME)*, Montreal, Quebec, Canada.

[2] Dannenberg, R. "Portmidi", Project home page http://www.cs.cmu.edu/~music/portmusic/ .

[3] Letz, S., et al. "Midishare", home page http://midishare.sourceforge.net/.

[4] Butenhof, D. R. *Programming with POSIX Threads*. Boston, 1997.

[5] Byers, G. "OpenMCL Common Lisp", home page http://openmcl.clozure.com/index.html.

[6] Rhodes, C. et al. "Steel Bank Common Lisp", home page http://sbcl.sourceforge.net/.

[7] Kawai, S. "Gauche – A Scheme Interpreter" http://www.shiro.dreamhost.com/scheme/gauche/ .

[8] Steele, G., *Common Lisp the Language, 2nd edition.* Digital Press, Wobern, 1990.

[9] Bielman, J and L. Oliveira. "CFFI – The Common Foreign Function Interface." http://common-lisp.net/project/cffi/.

[10] Keene, S. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison-Wesley, Reading, 1989.

[11] Wright, M. and A. Freed. "Open Sound Control: State of the Art 2003", *Proceedings of the International Computer Music Conference*, Montreal, Canada, 2003.

[12] Taube, H., "Common Music: A Music Composition Language in Common Lisp and CLOS*", Computer Music Journal, 1991.* 15(2): p. 21-32.

[13] Boehm, H., A. Demers, and S. Shenker, "Mostly Parallel Garbage Collection", *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, SIGPLAN Notices* 26, 6 (June 1991), pp. 157-164.