EFFICIENT DYNAMIC RESOURCE MANAGEMENT ON MULTIPLE DSPS AS IMPLEMENTED IN THE NEXT MUSIC KIT

David A. Jaffe

This paper was presented at the 1990 International Computer Music Conference in Glasgow, Scotland.

It is a rather technical paper about the internal implementation of the Orchestra.

INTRODUCTION

The NeXT Music Kit DSP resource allocation system was designed to maximize both flexibility and efficiency.

The flexibility is achieved by making nearly every part of the system dynamically configurable during performance. For example, the musician need not decide in advance how many voices of each synthesis technique he will be using.

The efficiency is achieved by minimizing unnecessary host-to-DSP communication. This leaves the DSP free to compute music samples at a maximum rate so that the greatest number of voices can run in real time.

These goals are realized by sharing and reusing resources whenever possible, resetting and freeing resources only when needed, minimizing expensive DSP instructions, vectorizing sample computation, and compacting memory when necessary.

BACKGROUND AND CONVENTIONS

This article begins with an overview of the levels of abstraction used in the Music Kit and proceeds to the details of resource management.

Some familiarity with the NeXT Music Kit and the NeXT DSP unit generator system is assumed. For a background in these subjects see "Overview of the NeXT Sound and Music Kits" (Jaffe and Boynton) and "unit generator implementation for the 56001" (Smith) as well as the NeXT Technical Documentation. Familiarity with object-oriented programming is also assumed.

We use the convention that objective-C class names begin with an upper-case letter and are in mixed case, e.g. "UnitGenerator". The syntax for used to show that an object "foo"

is sent a message "doIt:" with an argument "bar" is the following:

```
[foo doIt:bar];
```

LEVELS OF ABSTRACTION

The Music Kit allocation system supports four levels of abstraction. An application can "come in" at any level. This allows the Music Kit to be used for a wide variety of applications from musical performance to musical composition, from scientific signal generation to sound effects for games.

The levels are:

1. DSP assembly language unit generator macros and DSP data
2. UnitGenerator objects and SynthData objects
3. SynthPatch objects
4. SynthInstruments objects

In level 1, allocation is done at assembly time. In levels 2, 3 and 4, allocation is done dynamically at runtime, managed by an instance of the Orchestra class. Furthermore, the Orchestra class itself manages any number of DSPs as a single resource pool.

Level 1 -- The DSP unit generator code and data

The most basic DSP resources are DSP "unit generators" and DSP data memory. Unit generators macros expand at assembly time into DSP generating or processing modules with optional inputs and outputs. Each of the inputs and outputs is the address of a "patchpoint", a block of DSP data memory 16 samples long.

Unit generators are affected in real time by setting their "arguments". Arguments may be data or addresses. For example, an argument may be a patchpoint address from which an amplitude value is read, or it may be a constant frequency value, or it may be a table address.

Unit generator computation is done in 16-sample units. This improves the speed of the sample computation by a large factor because the expensive initialization and finalization of each unit generator is performed at only 1/16 of the sampling rate.

The NeXT machine is bundled with a set of DSP unit generator macros provided as source code.

Note that we use the term "unit generator" both for the DSP macro, the code invoked by

expanding that macro, and the instance of a UnitGenerator subclass. The context makes clear which is meant.

Level 2 -- UnitGenerator and SynthData objects

To use a unit generator macro in the Music Kit, it is first converted to a UnitGenerator subclass by the utility "dspwrap". Additionally, a subclass of this class is generated for each input/output memory space combination.

For example, for a one-input/one-output unit generator called "add2", dspwrap creates a UnitGenerator subclass called "Add2UG" and four Add2UG subclasses for the memory space combinations xx, xy, yx and yy. These are called "Add2UGxx", "Add2UGxy", "Add2UGyx" and "Add2UGyy". The unit generator designer then edits the Objective-C code for Add2UG, adding conversion from "human units" such as Hz. to "DSP units" such as oscillator table increment. The subclasses of Add2UG automatically inherit the argument-setting methods.

The designer may also supply several methods, -idleSelf, -runSelf, and -finishSelf, defining the meaning of corresponding UnitGenerator states "idle", "running" and "finishing". These states are defined as follows:

A newly allocated UnitGenerator is always idle. The idle state is defined as a "safe" state. That is, the UnitGenerator must not be writing its output anywhere. As a convenience, the special patch point "sink" is set aside for a UnitGenerator to write its output when it is idle. "Sink" is, by convention, never read. Therefore it corresponds to "nowhere". If a UnitGenerator adds its output directly to the output sample, it implements -idleSelf by adding zeros to the output stream. Thus it too follows the rule of having "no effect" when idle.

The precise definition of "running" and "finishing" is up to the UnitGenerator designer. The "running" status corresponds to a UnitGenerator in use, while the "finishing" status correspnods to a UnitGenerator that is "wrapping up". For example, UnitGenerators that function as envelope generators are sent the -finishSelf message by the controlling object when a noteOff is received.

Each instance of a UnitGenerator subclass corresponds to a piece of running DSP code. DSP data and patchpoints are represented by instances of the SynthData class. SynthData is used to store tables, delay lines, constants, etc.

Allocation and deallocation of UnitGenerator and SynthData instances is managed by the Orchestra class. Deallocated instances may not be referenced, as they may be freed by the Orchestra at any time. The allocation algorithm is quite involved and is explained later in this article.

The Music Kit provides a library of UnitGenerator subclasses including oscillators, filters, adders, delay units, etc. Unit Generators may be used on their own or they may be combined into SynthPatches.

Level 3 -- SynthPatches

UnitGenerators may be grouped together into larger units called "SynthPatches". Various SynthPatch subclasses implement various synthesis techniques. An instance of a SynthPatch subclass represents a monophonic sound-producing entity, corresponding loosely to a "voice" on a polyphonic MIDI synthesizer.

Like UnitGenerators, SynthPatches are allocated and deallocated from the Orchestra. A single SynthPatch instance is constrained to reside on a single DSP. If a multi-DSP patch is desired, it must be broken into parts, each of which resides on a single DSP. These parts then communicate through a special inter-DSP UnitGenerator (see caveat below).

Each SynthPatch subclass is defined by three types of information:

1. The UnitGenerator classes instances needed to create an instance of the SynthPatch subclass and (optionally) the order in which they run.
2. The connections between the UnitGenerator instances.
3. The manner in which incoming notes are converted into UnitGenerator argument updates.

For portability, 1 and 2 are specified as data in an instance of the PatchTemplate class. For flexibilty, 3 is specified as code in the implementation of several methods.

A SynthPatch subclass may produce instances of several "flavors", each represented by a particular PatchTemplate. For example, a SynthPatch class can support a "flavor" that has vibrato and one that does not have vibrato. This allows the use of DSP resources to be fine-tuned to the musical situation at hand.

Like UnitGenerators, SynthPatches have three states, "idle", "running", and "finishing". The state changes are specified in the subclass methods -noteEndSelf, -noteOnSelf:, and -noteOffSelf:.

A newly allocated SynthPatch is always idle. An idle SynthPatch has finished on never-run envelopes. It may contain UnitGenerators in any state. Its only requirement is that it be "safe" -- it must write its output to "sink".

A "running" SynthPatch has received a noteOn but no noteOff. A "finishing" SynthPatch

has received a noteOff but its envelopes are still in their release portion.

The Music Kit provides a library of SynthPatch subclasses including FM synthesis, wave table synthesis, and plucked string synthesis. SynthPatches may be managed by the application or they may be managed by SynthInstruments.

Level 4 -- SynthInstruments

SynthInstruments manage SynthPatches and do voice allocation, in a manner similar to a MIDI synthesizer. If a SynthInstrument is used, the SynthInstrument rather than the application does the allocation of SynthPatches from the Orchestra.

SynthInstruments are beyond the scope of this article. They are described briefly in the appendix.

TECHNIQUES THAT MAKE EFFICIENT ALLOCATION POSSIBLE

The algorithms that manage UnitGenerators, SynthData and SynthPatches follow a number of principles:

1. Resources are shared whenever possible.

When a resource such as a wave table is loaded, it may be published as a "shared resource". In this case, a reference count is automatically kept. An allocation request for the resource returns the resource that is already there. This saves DSP memory and the host-DSP communication overhead involved in loading tables.

2. Resources are reused whenever possible.

When a UnitGenerator or SynthPatch is deallocated, it is not actually freed. Rather, it is put on an available list for that class. Similarly, when the reference count of a shared object goes to 0, it is not freed, it remains on the DSP but is slated for possible garbage collection.

3. Resources are reset only as needed.

The UnitGenerators in a SynthPatch may remain connected, even when the SynthPatch is idle. Furthermore, the UnitGenerators may be left running. The SynthPatch subclass' -noteEndSelf method need only insure that the instance is "safe". For example, a SynthPatch subclass implementation of -noteEndSelf may be as simple as:

```
[output idle];
```

Furthermore, a low-level mechanism supresses commands to set a UnitGenerator argument to a value to which its previously been set. This is only safe to do if the DSP code of a unit generator does not itself modify the argument. Thus, each UnitGenerator class specifies which of its arguments can be optimized in this manner by implementing the method +shouldOptimize:.

4. DSP jump instructions are avoided.

UnitGenerators are loaded in-line, rather than as subroutine calls, and are constrained to be loaded contiguously on the DSP. This eliminates the expense of DSP jump instructions. UnitGenerators are constrained to be added in a stack (LIFO) fashion, allowing DSP resources at the top of the stack to be freed with no host-to-DSP communication. New UnitGenerators are simply loaded on top of the old ones.

5. Resources are always freeable.

If a deallocated UnitGenerator gets trapped at the bottom of the stack, DSP memory compaction is triggered to free it. A command is sent to the DSP to move the UnitGenerators that are to be compacted. Very little host-to-DSP communication is needed, since the UnitGenerators are moved rather than reloaded. The compaction requires no change to the unit generators' state, since all inter-unit generator communication is via patchpoints which are addressed indirectly via unit generator arguments. The only change necessary is to redo the relocation fixups, usually only one per unit generator.

This system eliminates the problem of memory fragmentation for program memory. Note, however, that only UnitGenerators and their arguments are compacted. Data memory is not compacted. Therefore, it is possible that data memory will become fragmented over the course of a performance, resulting in reduced availability of contiguous large blocks.

6. DSP timed messages are bundled.

On the lowest level, DSP commands time-stamped for the same time are bundled into units, thus saving some host-to-DSP overhead. The Orchestra method +flushTimedMessages causes all buffered settings to be sent. This message is sent automatically by the Conductor during a scheduled or MIDI performance. Any DSP commands sent in response to asynchronous input such as mouse clicks must be followed by +flushTimedMessages.

In release 2.0 and later, +flushTimedMessages is replaced with a +lockPerformance/+unlockPerformance pair.

SATISFYING AN ALLOCATION REQUEST

All allocation requests are made to the Orchestra class. The Orchestra class manages a set of Orchestra instances, each of which corresponds to a DSP. The Orchestra class proceeds to try and satisfy the request on the first of its set of instances. If this attempt fails, it proceeds to the next instance. It continues until all instances have been tried or the request is satisfied.

SATISFYING A UNITGENERATOR ALLOCATION REQUEST

The algorithm is described below, in a step-by-step fashion:

1. The Orchestra class passes control to the first Orchestra instance.

2. Check for deallocated UnitGenerator.

The Orchestra instance checks to see if there is already a deallocated instance of the proper class loaded on the DSP. If so, that instance is returned. (Note that the deallocated lists are actually maintained by the UnitGenerator subclass, thus eliminating the expense of finding the appropriate list.)

3. Try and find UnitGenerator in a deallocated SynthPatch.

The Orchestra instance searches its list of deallocated SynthPatches to see if any patch has the desired UnitGenerator. (When a SynthPatch is deallocated, its UnitGenerators remain allocated.) If one is found, that SynthPatch is freed, its UnitGenerators are deallocated and the found UnitGenerator is returned.

4. Pop UnitGenerator stack.

The Orchestra instance starts at the top of the stack and checks each UnitGenerator to see if it deallocated or in a deallocated SynthPatch. If so, that UnitGenerator (and its SynthPatch, if any) is freed. This process continues until an allocated UnitGenerator is found that is either in an allocated SynthPatch or not in a SynthPatch.

5. Try and create a new UnitGenerator.

If there are sufficient compute-time and memory resources left on the DSP, a new UnitGenerator is created and the new object is initialized and returned. Note that the UnitGenerator stack can spill off chip. In this case, a jump instruction to "leap" off chip is automatically inserted.

6. Compact UnitGenerator stack.

The Orchestra instance begins at the bottom of the stack and frees all deallocated UnitGenerators. It also frees all UnitGenerators in deallocated SynthPatches along with the patches themselves. Holes in the stack are squeezed out using memory compaction. The moved UnitGenerators are fixed up and the objects are sent the -moved message, in case they want to do any internal consistancy adjustment.
Synthpatches are also notified if one of their UnitGenerators is moved.

7. Try and create a new UnitGenerator.

If there are are sufficient compute-time and memory resources left on the DSP, a new UnitGenerator is created and the new object is initialized and returned, as before. Otherwise, control is passed to the Orchestra class.

8. Try the next DSP.

The Orchestra class repeats the above algorithm with the next of its Orchestra instances.

SATISFYING A SYNTHDATA ALLOCATION REQUEST

SynthData allocation is similar to UnitGenerator allocation with two exceptions:

* The deallocated SynthData is immediately freed. This keeps the DSP data heap as compact as possible. Note that no compaction of allocated data is done so the application need not worry about data moving unexpectedly.

* If the data can not be allocated because there is not enough memory, any shared data with a reference count of 0 is freed. This is done before any SynthPatches are dismantled.

SATISFYING A SYNTHPATCH ALLOCATION REQUEST

A Synthpatch request is defined in terms of the algorithms above. It proceeds as follows.

1. The Orchestra class passes control to the first Orchestra instance.

2. Check for deallocated SynthPatch.

The Orchestra instance checks to see if there is already a deallocated instance of the proper class and the proper PatchTemplate loaded on the DSP. To speed the search,

the address of the PatchTemplate rather than its contents are used as the search key. If so, that instance is returned.

3. Create a new SynthPatch.

A. Allocate first UnitGenerator in SynthPatch.

The first UnitGenerator in the SynthPatch's PatchTemplate is allocated from the first available DSP using the allocation algorithm described above. If this allocation request fails, the SynthPatch allocation request fails and the algorithm proceeds to step 4.

B. Allocate other UnitGenerators and SynthData in SynthPatch.

The allocation of the remaining UnitGenerators and SynthData proceeds, with these resources constrained to be on the same DSP as the first one allocated. If the allocation request fails, all UnitGenerators in the new patch are deallocated and the algorithm proceeds to step 4.

C. Connect UnitGenerators in new SynthPatch.

D. Initialize the new SynthPatch.

The new instance is sent the method -initialize. This method may be implemented by the subclass to do any special allocation or other behavior. If the initialize method returns a non-nil value, indicating success, the new SynthPatch is returned as the successful satisfaction of the allocation request. Alternatively, the subclass can abort the allocation by implementing its initialize method to return nil. In this case, the new SynthPatch is deallocated and the algorithm proceeds to step 4.

4. The algorithm proceeds to step 1 with the next Orchestra instance if any. If none, the allocation request fails and nil is returned.

CONCLUSION

These algorithms minimize host-to-DSP communication while retaining full dynamic capabilities, thus allowing the DSP to most efficiently compute samples. This benefit is passed on to the user in the form of more voices in real time and quicker non-real time turn around. Furthermore, the efficiency gained in host-to-DSP communication is not made at the expense of greater expense in the host process. On the contrary, by

reusing existing resources, the host actually does less work.

CAVEATS

The multi-DSP support is not yet working.

ACKNOWLEDGEMENTS

Many of the ideas described in this paper were worked out in conjunction with Julius Smith. The lazy garbage collection of SynthPatches and UnitGenerators is based on a system written by Bill Schottstaedt at CCRMA for the Systems Concept Digital Synthesizer. Thanks also to Andy Moorer for his design input.

APPENDIX 1 - THE SYNTHINSTRUMENT

In most musical applications, SynthPatch allocation is done not directly but via a SynthInstrument. The SynthInstrument may do its allocation in one of two modes: "automatic mode" or "manual mode".

In "automatic mode", patches are allocated from a global pool and shared dynamically among all SynthInstruments.

In "manual mode", allocation is done from a fixed local pool of patches. The term "manual" is used because the application deteremines how many patches are assigned to each SynthInstrument. Note that the number of patches managed by the local pool may be changed over time during a performance.

Any number of SynthInstruments may be active at once, limited only by the available DSP resources. The only restriction on the SynthInstrument is that it manages patches of only one type. A SynthInstrument can manage any number of "flavors" of the same Synthpatch class simultaneously.

SynthPatch premption is supported. Preemption is triggered when a note arrives for which no patch can be allocated. (In manual mode, this means there are no patches in the local pool. In automatic mode, this means the Orchestra cannot allocate a new patch.)

The first step in preemption is to select a patch to preempt. The default selection algorithm is based on time. A different method may be substituted by subclassing the SynthInstrument class and overriding a single method. The default algorithm is as follows:

If there are any finishing patches, the patch that earliest received the noteOff is selected for preemption. Otherwise, if there are any running patches, the patch whose noteOn was received the earliest is selected. Otherwise, the note is omitted.

Once a patch is selected, preemption proceeds as follows:

1. The patch is notified that it is being preempted. The SynthPatch subclass can specify whatever internal adjustments it wants by implementing the method -preemptFor:. A normal implementation of this method sends the -abort message to its envelope generators.

2. The pending new note is scheduled (with the Conductor, the Music Kit scheduler) to be sent to the preempted instance after a short delay. The delay allows the amplitude envelope of the old note to go to 0, thus preventing a click. The time defaults to .006 seconds but may be set differently.

3. After the delay, the noteOnSelf: method is invoked and the new note begins.

By allowing control over the preemption time, the Music Kit allows the application designer to optimize the trade off between fast response (small preemption time) and click-free preemptions (longer preemption time).