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

David A. Jaffe
NeXT, Inc.
900 Chesapeake Drive
Redwood City, CA 94063
(415) 366 0900
(415) 780 3715 Fax
David_Jaffe@next.com

ABSTRACT: The NeXT Music Kit DSP resource allocation system maximizes both flexibility and efficiency. The flexibility is achieved by making nearly every part of the system dynamically configurable during performance. Furthermore, an application can "enter" at various levels of abstraction, supporting such diverse uses as musical performance and composition, scientific signal generation, and sound effects. The efficiency is achieved by minimizing unnecessary host-to-DSP communication. Allocation algorithms emphasize the sharing and reusing of resources, the resetting and freeing of resources only when needed, the minimizing of expensive DSP instructions, the vectorizing of DSP computation, and the compacting of DSP memory when necessary.

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. Techniques invoked to realize the goals of flexibility and efficiency include 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.

This paper assumes a familiarity with the Music Kit. For introductory material, see the references listed at the end of the paper.

LEVELS OF ABSTRACTION

The Music Kit allocation system supports four levels of abstraction. An application can "enter" 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 four levels of abstraction are given below, from lowest to highest:

1. DSP assembly language "unit generator" macros (DSP processing/generating modules) and DSP data.
2. UnitGenerator objects and SynthData objects (basic Music Kit classes created from unit generator macros).
4. SynthInstrument objects (managers that route incoming notes to appropriate SynthPatch instances).

ICMC GLASGOW 1990 PROCEEDINGS
Level 1 is most useful when debugging a new unit generator macro (see Smith [3]). In levels 2, 3 and 4, allocation is done dynamically while playing music, managed by an instance of the Orchestra class. Furthermore, the Orchestra class itself manages any number of DSPs as a single resource pool. Most music performance applications use level 4, the most automatic level, in which SynthInstruments manage an allocation on a demand-driven basis, based on an incoming stream of performance data (Note objects). SynthInstruments are beyond the scope of this paper, which focuses primarily on the details of levels 2 and 3.

TECHNIQUES THAT MAKE EFFICIENT ALLOCATION POSSIBLE

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 maintained.

2. Resources are reused whenever possible. When a UnitGenerator or SynthPatch is deallocated, it is not actually freed, but merely slated for possible garbage collection. Similarly, when the reference count of a shared resource goes to zero, it is retained and 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. The UnitGenerators may even be left running. The SynthPatch subclass need only insure that an idle SynthPatch instance is “idle”, by writing its output to a special SynthData object.

4. DSP jump instructions are avoided. UnitGenerators are loaded in-line, rather than as subroutine calls, and are loaded contiguously on the DSP, eliminating computationally expensive jump instructions. To make this possible, DSP code is stored in relocatable form and direct address references are “fixed up” when the UnitGenerator is loaded.

5. Unused resources may always be freed. UnitGenerators are loaded as a stack. If a deallocated UnitGenerator gets trapped at the bottom of the stack, the stack is compared to free. Very few host-to-DSP communication is needed, since the UnitGenerators are moved rather than reloaded. The compaction of a UnitGenerator requires no change in its state, because all communication between UnitGenerators is done with a level of indirection. Thus, only the relocation fix-ups need to be updated.

6. DSP timed-messages are bundled. DSP commands that are time-stamped for the same time are bundled into units, saving host-to-DSP overhead. In addition, duplicate commands to set UnitGenerator DSP arguments are suppressed when it is safe to do so.

7. DSP computation is recurved. UnitGenerator computation on the DSP 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 UnitGenerator is performed at only 1/16 of the sampling rate.

HOW AN ALLOCATION REQUEST IS SATISFIED

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

We now give the algorithms for satisfying UnitGenerator and SynthData allocation requests. Note that these algorithms are performed automatically...
1. The Orchestra class passes control to the first Orchestra instance.
2. Check for a deallocated UniGenerator ("UG") of the requested type. If found, return it.
3. Look for a UG of the requested type in a deallocated SynthPatch. If found, free the SynthPatch and return the UG.
4. Beginning at the top of the UG stack, free all idle UGs as well as UGs in idle SynthPatches. Continue until a non-idle UG or UG in non-idle SynthPatch is found.
5. If there is sufficient DSP memory and compute time to accommodate a new UG of the requested type, create and return the new UG.
6. Compact the UniGenerator stack and repeat step 5. If failure occurs to step 7.
7. Repeat the algorithm with the next Orchestra instance (i.e., DSP). If no more, return "nil".

SynthData allocation algorithm

SynthData allocation is similar to UniGenerator allocation with two exceptions: A deallocated SynthData is immediately freed to keep the data heap as compact as possible. Also, if there is not enough memory to allocate the new SynthData, any shared SynthData with a reference count of zero is freed.

SynthPatch allocation algorithm

A SynthPatch request is defined in terms of the two algorithms above. It proceeds as follows.
1. The Orchestra class passes control to the first Orchestra instance.
2. Look for a deallocated, SynthPatch of the requested type. If found, return it.
3. Create a new SynthPatch of the requested type.
   A. Allocate the UniGenerators and SynthData that comprise the new SynthPatch, using algorithms given above. If failure, go to 4.
   B. Connect the UniGenerators in new SynthPatch.
   C. Initialize the new SynthPatch. If failure, go to 4.
   D. Return the new SynthPatch.
4. Repeat the algorithm with the next Orchestra instance. If no more, return "nil".

SUMMARY

These algorithms provide the flexibility of dynamic allocation while minimizing host-to-DSP communication, thus allowing the DSP to most efficiently compute samples and minimizing the load on the host. The benefit to the user is more synthesis power in real time and quicker turn-arounds when working out of real time.

ACKNOWLEDGEMENTS

Much of this paper was worked out in conjunction with Julian Smith. The lazy garbage collection of SynthPatches and UniGenerators is based on a system written by Bill Schroeder at CCRMA for the Systems Concept Digital Synthesize. Thanks also to Andy Mower for his design input.

REFERENCES

4. NoXt Technical Documentation.

ICMC GLASGOW 1990 PROCEEDINGS