This paper describes the fundamental design of PWGL-Synth 2, a revision of the realtime audio DSP component of the visual PWGL language. The design focuses on exploiting hardware parallelism to improve the performance of the system. Highly general principles for scheduling concurrent processing in the context of music DSP are examined. These concepts can be leveraged to utilize both multiple processors and within each processor, any vector processing capabilities available. PWGLSynth 2 is implemented as a multi-threading, vectorizing compiler, directly producing native machine code, in contrast to PWGLSynth 1, a traditional interpreter.

1. INTRODUCTION

Parallel processing has recently become an increasingly important topic in computational research. While designing and implementing serial algorithms and programs is traditionally considered easier, hardware issues have forced a shift into multiple processing core paradigms.

This paradigm shift introduces new design challenges and optimization problems to software design. As the field of sound synthesis and audio analysis can utilize highly processor intensive algorithms, computational efficiency is one of the central design problems of a computational platform. As the trends in computing hardware dictate on multiple levels, any forward-looking computational platform design is highly motivated to utilize multiple processing cores and any inherent explicit parallelism these cores in themselves offer. One such computational platform is PWGLSynth 2, a new version of the audio synthesis extension to the visual programming language PWGL[2].

In the Section 2 of this paper we will provide motivation and goals for the redesign of our synthesizer. In the Section 3, some key principles of concurrent programming in audio DSP are presented. The implementation issues of these principles are discussed in the Section 4. Finally, a case study of leveraging these principles in the context of PWGLSynth 2 comprises the final Section 5, showing how one of the fundamental design problems of modular audio systems, buffering, is naturally resolved via the emergent properties of the system.

2. TOWARDS PWGLSYNTH 2

PWGLSynth 2 is a research project to implement a novel, highly general audio DSP component for PWGL. In total, our system allows the user to program musical tasks from algorithmic composition to notation and synthetic audio. The main syntax of PWGL consists of connecting boxes to each other via patch connections. The previous version, PWGL-Synth 1, has been used to expressively perform musical scores on physics based instrument models.

The main goals of PWGLSynth 2 are to improve performance, especially in the mixed multirate signal case, essential in the case of audio analysis, as well as generally, by taking advantage of vector processing capabilities of modern computer cores and utilizing multiple cores simultaneously. Various components of PWGLSynth 2 are in development. In addition, the authors strive to improve on the syntactic elegance and simplicity of the system.

2.1. The Unified Signal Model

The syntactic goal of PWGLSynth has always been to reduce the size of the kernel set of primitive operations or boxes required to build patches. This improves the expressiveness of the language as well as mitigates the learning curve. To pursue this goal, we have abolished the separation of audio and control rate signals, which in some systems require separate versions of otherwise very similar primitives.

Since control rate signals offer a significant performance benefit, in PWGLSynth 1 we provided a refresh mechanism for a box to treat one of its inputs as a control signal. Such an input receives notifications about signal changes, or a low frequency update tick, that enables it to respond to signal changes without continuously polling the input. While adequate for a simple case of control signals, this scheme is far from optimal in the general case.
2.2. The Functional Reactive Paradigm

The new control system being developed by the first author of this paper is based on the paradigm of functional reactive programming[5]. By formulating the DSP operations required to compute patch output as a functional expression, it is possible to ascertain which parts of the expression must be re-evaluated when a particular state of the system changes. This unified signal model eliminates the need to treat audio and control signals differently on patch level.

The patch is analyzed as a network of states, connected by functional expressions. These states may be inputs or outputs of the system, such as audio signals or user interface elements. Updating a state triggers the re-evaluation of all the states downstream from itself. Thus, a side branch of a patch can lie dormant when re-evaluation is not needed. The problems of optimal control signals have been discussed by the authors [3].

3. GENERAL PRINCIPLES OF PARALLEL EVALUATION

As the redesign of PWGLSynth coincides with the hardware shift to parallel processing, it is natural to consider methods of leveraging that trend. The central component of the synthesizer is the evaluator component, responsible for performing the computations required by the patch in a programmatically correct order. The main difference between straightforward, serial evaluators and parallel evaluators is the enforcement of scheduling rules.

3.1. Scheduling Rules for Serial Evaluators

A serial evaluator, such as PWGLSynth 1, must schedule a stream of operations in some particular, programmatically correct order. In a visual language, correctness can be demonstrated by showing that operations in the upstream of the data flow are performed before those that follow them.

There are typically many different schedules that are correct. The precise choice of a certain schedule over another one is usually not of great importance, as long as correctness is guaranteed. In serial evaluation, each operation is performed exactly once, one after another, and computational cost doesn’t change significantly when the processing order is altered. For the example in Figure 1, a typical serial schedule for the boxes shown would be A, B, D, C, E.

3.2. Scheduling Rules for Parallel Evaluators

On the other hand, when one wishes to exploit hardware parallelism, the situation becomes more complex. The fundamental problem of parallel algorithm design is determining the parts of the algorithm that can be executed concurrently. If parallelism is not fully exploited, some computational resources in the system may be idle for a significant portion of the evaluation, not contributing to the evaluation performance. The same basic scheduling principle still holds for parallel evaluation: the scheduler must guarantee that the order of data flow is preserved.

3.3. The Least Restrictive Set of Scheduling Constraints to Enforce Correctness

Let us examine the concept of the available operation pool. When the evaluation of an algorithm is started, there is a number of operations that are ready to be executed right away. These are the leaves of the expression tree, the outer extremities of the visual patch. Since they have no dataflow upstream, there are no dependencies. Any or all of these operations could be immediately executed.

When one of these operations are completed, the operations immediately downstream from them are examined. Subsequent operations that depend on already completed operations are in turn added to the pool. Thus, the available operation pool represents the parts of the algorithm that are ready for evaluation. The operations in the pool include all the possible choices for proceeding in programmatically correct order. This is the least restrictive set of constraints for correct scheduling.

For the example in Figure 1 the initial available operation pool would consist of A, B and C. Once A and B are evaluated, D is added to the pool. Only when A, B, C and D are all evaluated, E will become available. A further important point is that all the operations in the pool are always independent of each other. Therefore they can be executed in any arbitrary order or even concurrently.

4. PARALLEL COMPUTATION IN PWGLSYNTH 2

4.1. Thread Level Parallelism

To exploit multiple computation cores, the patch must be divided into a number of localized subtasks that can be executed in parallel. PWGLSynth 2 utilizes the well known concept of tasks and worker threads. To reduce multithreading overhead, a group of operations that form a subexpression of the total patch are grouped into a task. Such a task is
the work unit dealt out to a particular computational core at a time.

4.1.1. Division of labor: tasks

The evaluator analyzes the patch, building work units known as tasks from independent, local groups of operations. The concept available operation pool concept can be applied to tasks as well. This time the tasks, each consisting of multiple operations, are the units in the pool. Once all tasks upstream of a particular task are completed, the task itself is put into the pool.

The task processor consists of worker threads that keep polling the pool for available tasks. If one thread per computational core is created, each core can potentially be used in full. The concept of job queue with worker threads is widely used in concurrent computing. JACK, the inter-application audio server, utilizes a similar model on application level [4].

4.2. Instruction Level Parallelism

Modern computer hardware offers two kinds of parallelism on the instruction level, right down to the individual primitive operation carried out by the processor. Implicit parallelism is extracted from the program automatically by the processor. This kind of parallelism is easy to exploit since it is performed transparently by the hardware.

4.2.1. Explicit parallelism

It is more difficult to leverage explicit parallelism, in which the compiler directly instructs the hardware to perform certain tasks in parallel.

The prevalent explicitly parallel paradigm today is Single Instruction Multiple Data, or SIMD. This form of explicit parallelism is found in general purpose computing architectures such as the x86 and PowerPC as well as specialized hardware like Graphics Processing Units and Stream Processors.

The main constraint of SIMD-type parallelism is that the parallel operations must be of the same type, albeit with different operands. The problem of producing SIMD-parallel code from the available operation pool is therefore reduced to grouping together bundles of similar operations that are simultaneously available.

4.2.2. PWGLSynth 2 as a just in time compiler

One of the main design decisions for PWGLSynth 2 was to change the lowest level of the evaluation model from interpretation to compiled native code. At synthesizer startup, the evaluator compiles the patch into a set of subroutines which are used in the actual audio processing. By producing machine code for use in the audio loop, the evaluator essentially becomes a Just in Time compiler [1]. A chunk of machine code is produced for all update routines of states in the system. When a state of the system needs to be updated, as described in Section 2.2, the evaluator simply calls the corresponding generated subroutine.

5. OUT OF ORDER EVALUATION AND BUFFERED AUDIO

Finally, we consider the impact of parallel evaluation on another method of increasing audio processing efficiency; processing audio in chunks of several samples instead of sample at a time.

In PWGLSynth 1, buffering was rejected because one design specification of the system was to provide recursive connections with unit delay, the minimum possible in a digital system. This specification is very important in physics based models, one of the main applications of PWGLSynth 1. However, such recursion is still pretty rare, and sometimes it would be clearly beneficial to opt for buffered processing. While it would be possible to implement a serial evaluator capable of separating portions of the patch for sample-by-sample or buffered processing, the complexity of the scheduling algorithm would greatly increase.
5. OUT OF ORDER EVALUATION AND BUFFERED AUDIO

Finally, we consider the impact of parallel evaluation on another aspect... we notice that this is completely valid as long as there is no recursion. Why this is so will become evident in the following paragraphs.

Figure 3 is an illustration of how the resulting patch would look on the low level. This illustration is not meant to convey patch level user experience, as any buffering should be made transparent to the user. The buffer size implied here is four samples, smaller than in practical reality. The patch here is shown duplicated four times, each version referring to a different input and output samples of buffered input and output. Unit delays now manifest as connections breaking the time hierarchy, referring to their left side siblings on the time vector.

What actually happens in buffered processing of audio is out of order evaluation. Some operations get to process the complete time slice represented by the buffer before other operations can even get started. If buffered processing is considered from the point of view of the available operation pool, we notice that this is completely valid as long as there is no recursion.

Figure 3. An illustration of a potentially parallelizable implementation of the filter displayed in Figure 2.

If a recursion is encountered in the patch, scheduling rules will automatically prevent further parallelization of the patch in the recursive section. An example of a minimal recursive filter along with a parallelization attempt is shown in Figure 4.

Figure 4. When the parallelization method is tried on the recursive filter, data flow dependencies prevent parallel processing in the feedback section.

As can be seen in Figure 4, the result of each filter computation is needed for the next sample frame. This is the very definition of unit delay recursion. The fundamental scheduling rules prevent any out of order or parallel evaluation of the filter for the recursive section of the patch, as doing otherwise would violate data flow integrity.

This indicates that buffering or per-sample processing emerge spontaneously from the scheduling rules given as the minimal set of constraints for evaluation order correctness. All in all, buffering can be viewed as yet another application of efficient ordering of operations.

6. CONCLUSION

This paper presented the fundamental design of PWGLSynth 2. The system aims to a dramatic performance improvement over PWGLSynth 1 by leveraging concurrent processing and moving from interpreting patches to compiling them into native code. At the same time, the syntax and usability of the system will be simplified.

Principles for concurrent evaluation were examined. The minimal set of constraints for correct scheduling was examined and found suitable for parallelizing computation of audio DSP patches on multiple levels; on thread level, dividing the work among multiple computational cores, and on instruction level, taking advantage of vector processing capabilities. The use of buffering in audio processing was shown to be an emergent property of an advanced evaluator.

PWGLSynth 2, a system exhibiting these features, is in development. We aim to provide a free, functional technology preview as an optional PWGL component during the year 2009, gradually working towards replacing PWGLSynth 1 entirely.

This work has been supported by the Academy of Finland (SA 105557 and SA 114116).

7. REFERENCES


