A Run-Time System for Arctic

Roger B. Dannenberg
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA
email: roger.dannenberg@cs.cmu.edu

ABSTRACT: A run-time organization has been designed and implemented to support the high-level, functional, real-time language Arctic. The run-time system automatically determines the execution order of expressions in order to obtain signal propagation from inputs to outputs in a single sample time. Signal values are cached to avoid redundant computations when one expression is used in several places. The object-oriented run-time system also supports multiple parallel instantiations of prototype behaviors, which can change signal computations dynamically.

Introduction. Arctic is a very high-level, functional language for real-time control. Previous papers have described Arctic [1, 2, 3], documented a non-real-time implementation of Arctic [4], and shown how Arctic semantics can be applied to the domains of composition [5] and signal processing [6], but there has never been a real-time implementation of Arctic. An important step toward this goal is the design and implementation of a run-time system that supports the instantiation of Arctic prototypes, scheduling of events, and ordering of execution evaluation to be consistent with data dependencies.

Arctic differs from procedural programming languages in that most Arctic expressions represent signals that are computed over some interval of time. Arctic variables may be assigned only a single value (a signal), and multiple instances of Arctic prototypes may be instantiated to run in parallel. The fact that Arctic values are signals leads to a system in which expressions are evaluated periodically to generate a stream of samples. The run-time system must provide a representation for these computations and simulate parallel execution by interleaving the computations of many parallel signals.

Arctic is intended as a language for describing control signals such as pitch and envelope functions that control a digital signal processor. This is a domain where the power of a very high-level language is most important. A typical application for an Arctic program is to describe the processing that links a set of samples on an electronic instrument to a set of synthesis algorithms that produce sound. Arctic can also express the computation of audio signals directly, but ordinary processors are not fast enough for this approach.

Evaluation Order. At any given instant during program execution, an Arctic program consists of some number of active instances of assignment statements that relate variables to expressions. For example, the following statements might be in effect:

\[ A := B \cdot \text{ramp} \cdot 5; \]
\[ B := X \cdot 3; \]

Here, A and B are variables, X is an external input, and \text{ramp} is an Arctic prototype that generates a linear ramp function from 0 to 1. At run time, assignments are re-evaluated at every sample time, and the order of evaluation is such that variables are defined with up-to-date values before they are used. At any given sample computation, the second assignment must be evaluated to get...
the current value for \( B \) before the first assignment, which uses the value of \( B \), is performed. Failure to evaluate expressions in the correct order increases latency. In this example, if the top expression were evaluated first, it would see the previous value of \( B \), adding an extra sample time of latency from input \( x \) to output \( y \).

It turns out that there is no simple static analysis of the program that can determine the run-time order of evaluation because new assignments can come into existence at run-time, creating new dependencies, and altering the necessary evaluation order. Arctic expressions are translated into instances of objects in the object-oriented run-time system. These objects respond to the "access" message by computing the Arctic expression.

For example, the object for \( A \) above responds to the "access" message by sending "access" to both \( B \) and \( C \) and combining the results to form \( B \times C \). This guarantees that up-to-date values will be used for both \( B \) and \( C \). For efficiency, when an expression is evaluated, its value and a timestamp are saved in the object so that another "access" at the same logical time (sample number) does not have to re-evaluate the expression. At the beginning of every sample computation, the global time is advanced, invalidating the saved values and forcing a re-evaluation of all expressions upon the next "access".

Instances of Prototypes. Another feature of Arctic is the idea of prototypes; that is, events give rise to instances of prototype behaviors. For example, the following prototype implements the response to a keydown event, and multiple keydown events will give rise to multiple instances of the keydown prototype:

```plaintext
keydown(in pitch, in vel) causes
  osc(pitch + vibrato, amp_envelope * vel);
```

To support multiple instances, prototypes are represented as classes in the object-oriented run-time system. Whenever an event occurs, an instance of the corresponding class is created. In many cases, the initialization method of the class will create other classes; for example, the keydown class shown above would create instances of the prototypes vibrato, amp_envelope, and osc, resulting in an interconnected network of objects that perform the required signal computations.

Performance. At present, the run-time system runs on a 16MHz 68020-based Commodore Amiga personal computer with a Motorola 68881 floating point chip. Arctic programs must be hand-compiled into C with object-oriented extensions. The run-time system consists of a small library of support functions and a library of classes that are specialized to form a set of subclasses specific to each Arctic program. Our objective was not to produce a practical and complete implementation, but merely to explore and measure the performance of a run-time system that can support real-time Arctic execution.

All Arctic expressions are computed in floating point representation. If the output instructions are removed along with "wait" instructions that synchronize samples to a real time clock, then a typical Arctic program performs 12,800 "access" methods per second. Of these, about half retrieve cached values and the other half result in the re-evaluation of the associated expression. Instantiation of objects requires an average of about 660\(\mu\)s, but it is common for the response to an event to result in the creation of a dozen or more instances. The implementation seems fast enough to implement the interface between a controller and a DSP-based synthesizer, but not fast enough to control multiple voices in real-time. Optimization of the code could result in a factor of 2 or 3 improvements, and of course faster processors are also available.

By keeping track of dependencies among expressions, the Arctic run-time system minimizes the latency between inputs and outputs. The run-time ordering of expression
evaluation guarantees that the expressions will be evaluated such that a change in an input propagates through all expressions and reaches the output(s) in a single sample. This property is obtained in other systems such as Forms [7] and Formula [8] only by placing mutations on signal dependencies and by manually specifying the order of evaluation.

The nesting level of "access" method invocations was found to be as high as 5, even in simple programs. This should not be taken as a typical number but just an indication that expression ordering has a significant benefit. Since values propagate through at least 5 expressions from input to output, latency is substantially reduced by evaluating expressions in the order that minimizes latency. In Arctic, the latency is always less than two sample times (one to synchronize with incoming data and one for computation).

Conclusions. We have implemented a run-time system for Arctic that supports the dynamic ordering of expression evaluation to achieve low latency. The system also supports multiple instances of behavior prototypes. Measurements have demonstrated that the system is fast enough to support complex real-time control systems for at least a single instrument implemented by a DSP.

Space limitations prevent a full discussion of the run-time system, which includes an interesting technique for dynamic program optimization in which expressions that become constant functions can be replaced by constants. The interaction between events and signals is also interesting; these topics will be discussed in a future paper.

The next step in this development could be a compiler that translates Arctic into C for execution within the run-time system. Since the run-time system is object-oriented, it also seems reasonable that some of the techniques proposed for Arctic could also be applied to more conventional languages or made available as software libraries. This would simplify the task, as does Arctic, of programming highly dynamic, real-time systems that include both event-based logic and the computation of sampled control signals.

Acknowledgment. The author is grateful to Yamaha Music Technologies for their support of this work. Hal Mukai has contributed to the initial implementation effort.

References


