Csound: Parser and Engine Technical Documentation

Version 1.0 - 2007.11.xx

Steven Yi

Table of Contents

1 Introduction

This document evaluates the current design of Csound's parser/compiler and runtime engine, discusses the merits and limitations of the current design, as well as proposals for changes to achieve the growing set of new requirements to keep Csound viable into the future.

2 Parser

2.1 Overview

Csound has historically used a handmade C parser and compiler for reading in Csound ORC code and compiling the data structures that define instruments, user-defined opcodes, and other data types which Csound later uses at runtime for sound processing. Over the years the compiler has been extended in ways to handle new language features, though doing so with a handmade parser has not come without a cost in complexity of understanding the code base and consequently a difficulty in maintaining the current parser. The following sections will discuss the current Csound parser/compiler design, issues and capabilities desired for a new parser/compiler, and proposed changes to achieve those desired capabilities.

2.2 General Parser Design

2.2.1 Current Parser Design

Csound's current parser and compiler are currently coupled into a single pass design. The lexer generates stream of TEXT tokens which the otran function reads one at a time and compiles at parse time into the data structures necessary for Csound at runtime.

Lexer

The current Csound lexing and parsing functions are found in Engine/rdorch.c. The implementation of rdorch is currently of a monolithic nature. (NEED TO SEE IF PREPROCESSOR IS DONE IN LEXER OR PARSER).

Parser/Compiler

The current Csound parser/compiler functions are found in Engine/otran.c. The otran function reads a stream of TEXT tokens and parses/compiles them into runtime data structures with no intermediary data representation (i.e. Abstract Syntax Tree). The design of otran is also currently of a monolithic nature.

When compiling INSTRTXT data structures, the compiler currently finds names for variables and within a map keeps a record of the name of the variable and the index of that variable name. When the translation pass has run through the instrument, it goes and counts the number of the various types of variables found (i-, k-, a-, w-, p-, S-, etc.), figures out how much memory the sum of those would take and allocates one giant block of memory. Afterwards, the memory is divided up into parts equal to the size of each variable type, then assigns to each variable found in the instrument the memory address each variable is to point to within the giant block of memory. The address of the giant block of memory is used later when instances of an instrument are allocated and a single memcpy can quickly and efficiently create a new instance of an instrument.

The most important aspect to note about the current compiler scheme for memory allocation is that at the end of compilation, the variable name map for both the local instrument and the global

context for global variables is discarded, leaving further possibilities of modification of the instrument impossible as retrieval of variable names is impossible. The giant block of memory approach, while extremely efficient in terms of both prevention of memory fragmentation as well as speed, prevents runtime instrument modification.

2.2.2 New Parser Design

The New Parser is based on using GNU Bison and GNU Flex to create a formal lexer and grammar. By defining the language with these tools, the language of Csound becomes easier to understand within the context of well-understood language definition schemes. By relying on the well-tested lexing and parsing technologies, problems with parsing and lexing should only arise on higher levels of language parsing and not on lower levels. This should also open up review and experimentation with the Csound language to those who are familiar with these language tools and not just those familiar with the Csound codebase.

The New Parser design also separates parsing into distinct passes. In the older handmade parser, many things like verification and optimization were done inline with the parsing and compilation pass. While perhaps offering some degree of efficiency, this also limits maintenance greatly. By separating into distinct passes for initial parsing into a parse tree, tree verification, tree optimization, and final compilation, the unique concerns can be applied at each step explicitly and verification rules and tree transformations can be cleanly applied. It is hoped that modern compiler techniques for code optimization (such as those based on SSA) can be applied to the Abstract Syntax Tree (AST) used in the New Parser.

Also, as part of the the New Parser work, for better separation of the parser and the compiler, a suite of new compiler functions for creating the data structures Csound uses at performance time have been developed[1]. This is a requirement to compile from the TREE structure which the new parser creates which is not compatible with the TEXT token stream that the old parser used. The creation of new compilation functions has been a good opportunity to refactor and breakdown the monolithic otran function into smaller, more coherent and easily understood functions.

2.3 User Defined Opcodes

User-Defined Opcodes (added to Csound by Istvan Varga) are built on top of the named instruments feature (added to Csound by Matt Ingalls) where one is able to call UDO code as if it was a normal opcode.

Underneath the hood, the UDO is added as an instrument to the INSTRTXT chain held in the CSOUND data structure. When parsed and compiled, UDO's are handled in the following order:

1. When initial UDO definition opening is found ("opcode xxx, i, i"), an opcode entry is immediately added with that signature. This needs to be done when this opcode definition is first found in case the UDO is recursive and needs to call itself. If it is recursive, when the parser finds the call to itself it needs to identify that name of the opcode as a call to an opcode which requires the entry in the opcode list.

1 The New Parser reuses a number of compiler functions from the old parser. These functions are likely to be replaced with different functions however when the data structure for instruments and other things are changed to be atomically modifiable and capable of being updated at runtime, which will require a new method of memory allocation when compiling and allocating instruments. The runtime engine are discussed in the "New Engine" section.

2. In the old parser, building of the INSTRTXT and OPTXT chains were done while

3 New Engine

3.1 Overview

The current Csound engine is highly optimized for memory and speed, yielding one of the fastest and optimized general-purpose synthesizers currently available. However, these optimizations come at a cost in that certain capabilities which are desired by the community (such as runtime modification of instruments) which have arisen over time are not possible without redesigning the Csound engine and data structures. The following sections will discuss the current Csound engine design, issues and capabilities desired for a new engine, and proposed changes to achieve those desired capabilities. Besides those issues regarding runtime modification, other data structure design requirements will be discussed which are necessary for multithreaded engine support.

3.2 General Design

3.2.1 Current Design

3.2.2 New Engine Design

Memory Allocation

4 Designing for Multiprocessor Systems

The current target system for multiprocessor support in Csound in multiple-processor, shared memory systems such as multi-core Intel and AMD processors. Other types of systems not currently targeted are multiple-processor, non-shared memory systems like those based on computing clusters and systems built on MPI.

4.1 Csound Design Changes

Multicore software development

4.2 Required Changes to Csound for Multicore

The following are some notes on identified points of change required to handle multicore support.

Removal of csound->currevt

The current memory structure for INSTRTXT creates a singly-linked linked list to represent the opcode chain as a flat list. The engine is currently designed to assign the csound->currevt to be the head of the current instrument event opcode chain and this is used when searching for labels within the code. This is because the chain is only singly-linked, the code that searches for labels looks at the head of the chain and iterates through until it finds the label to direct control flow to. However, by having only a single currevt at a time, this makes it impossible to multithread running of instruments.

The proposed solution is to remove use of currevt and instead make INSTRTXT's be doubly-linked lists. Opcodes which redirect control flow to labels can then use the backwards link to first go to the head of the chain and then interate forward from there. This will eliminate the necessity for currevt..