Платформа для машинно-независимого исполнения программного кода с возможностью JIT-компиляции

Целью настоящего проекта явилась разработка простой и удобной в использовании платформы для машинно-независимого исполнения программного кода в виртуальном процессоре (с возможностью JIT-компиляции в машинный код центрального процессора).

Основными задачами разработки являлись обеспечение кроссплатформенности, абстракции от входного языка, возможности работы с кодом в формате обобщённого промежуточного представления и создание простого минималистичного клиентского API.

Ядро платформы выполнено с использованием понятия байт-кода и поддерживает сериализацию, десериализацию и компоновку загруженных данных в любых комбинациях. Виртуальный процессор был построен по принципу стековой машины с набором команд, оптимизированным для ручного написания кода; также поддерживается работа с регистрами, памятью и стековым фреймом. JIT-компиляция была реализована для архитектуры Intel IA-32E.

Работа выполнена на языке C++ (стандарт C++11) без использования каких-либо библиотек помимо стандартной. Это и позволило обеспечить кроссплатформенность проекта на уровне исходного кода наряду с достаточно высокой скоростью работы.

Ядро

Платформа имеет конвейерное устройство и разработана в соответствии с принципами модульности и расширяемости. В её состав входят:

- уровень абстракции от ОС
- ядро, определяющее конвейер и межмодульные интерфейсы
- набор взаимодополняющих модулей, каждый из которых реализует определённый функционал и предоставляет ядру соответствующий интерфейс

Ядро платформы содержит реализацию клиентского API вкупе с основными определениями и межмодульными интерфейсами. Оно определяет типы данных и такие понятия, как состояние процессора, контекст (самостоятельная единица исполнения) и байт-код. В частности, в ядре определены структуры данных, составляющие байт-код — инструкция, элемент данных, символ и ссылка. Платформа поддерживает два типа данных: целочисленный и вещественный.

Также в ядре предусмотрена возможность взаимодействия с клиентским приложением (интроспекции). В частности, допустимы любые операции с контекстами и его содержимым; также поддерживается изменение и/или частичное переопределение командного набора виртуального процессора.

Состояние виртуального процессора фактически расположено в ядре и является глобальным для каждой конкретной копии платформы в памяти. Оно состоит из набора общих регистров, командного указателя и регистра флагов. Регистры процессора являются универсальными и могут содержать значение любого типа данных (в каждый момент времени — только одного).

Байт-код, MMU и контексты

Как уже было упомянуто ранее, ядро функционирует в соответствии с конвейерным принципом. Конвейер состоит из четырёх шагов: загрузка исходного текста, сохранение частичного образа байт-кода в отдельный контейнер, компоновка байт-кода и его исполнение в виртуальном процессоре. На последнем этапе вместо такого исполнения возможна передача контекста в модуль платформо-зависимого исполнения (JIT-транслятор), становящийся ответственным за все дальнейшие операции (применяется принцип чёрного ящика).

Байт-код, полученный в результате считывания одного исходного файла, помещается в так называемый контекст исполнения, являющийся приблизительным аналогом задачи в терминологии x86. За хранение контекстов отвечает модуль MMU. Каждый контекст представлен в платформе совокупностью секций кода, данных и символов вкупе с некоторыми метаданными (например, адресом точки входа). Отличие контекстов от задач x86 состоит в том, что стек данных и состояние виртуального процессора обобщены между контекстами. В целом, благодаря наличию такого промежуточного представления кода возможна полная абстракция от языка исходного текста. Также контексты подлежат сериализации в двоичные файлы, по структуре схожие с объектными файлами.

В отличие от состояния процессора, определённого на уровне ядра платформы, доступ к стеку производится через интерфейс модуля MMU. Архитектура платформы предусматривает наличие отдельного стека для каждого типа данных. При этом ячейки секции данных, как и регистры платформы, могут хранить значения любого типа. Стоит отметить, что виртуальный процессор осуществляет проверку типов при обращениях к данным. Также байт-код платформы поддерживает сложную адресацию, а именно есть возможность адресации со смещением и разыменования.

После создания контекстов отдельный модуль производит их компоновку, которая может происходить с объединением нескольких контекстов. Благодаря этому, становится возможна раздельная компиляция и, в частности, одновременное использование разных языков программирования. Объединение нескольких контекстов использует механизм релокации. Релокация заключается в сдвиге секций одного из контекстов на смещение, определяемое размерами второго контекста; тогда сама компоновка производится путём "наложения" секций друг на друга и слияния таблиц символов.

Исполнение

Первым этапом исполнения контекста является выбор способа исполнения, осуществляемый ядром для каждого контекста.

Стандартным способом является стековая виртуальная машина, исполняющая байт-код платформы. При исполнении этим способом ядро загружает инструкции и передаёт их в модуль логики исполнения, где инструкции декодируются (определяется их тип и загружаются операнды). Код операции вкупе с операндами передаётся исполнительному блоку требуемого типа данных, который непосредственно совершает действие инструкции. При этом исполнительный блок обращается к остальным компонентам платформы через модуль логики исполнения, а не напрямую.

Гибкое устройство виртуальной машины позволяет программно управлять исполнением (в т. ч. осуществлять отладку и трассировку) с помощью переопределения соответствующих методов в модуле логики исполнения. Также предусмотрена возможность взаимодействия управляемого кода и клиентского приложения через так называемые системные вызовы, представляющие собой пронумерованные процедуры, вызываемые из управляемого кода.

Необходимо заметить, что производительность не является основным требованием, предъявляемым к виртуальной машине, и, в том случае, если скорость исполнения недостаточно высока для конкретной задачи, возможно использование JIT-транслятора для построения машинного образа и исполнения кода непосредственно на центральном процессоре.

JIT-транслятор

JIT-транслятор, в рамках работы реализованный для семейства центральных процессоров Intel IA32E, служит заменой виртуальному процессору для тех случаев, когда требуется высокая скорость исполнения кода. В трансляторе используется собственный кодогенератор, реализующий System V x86_64 ABI (конвенция вызовов современных UNIX-подобных систем).

Для простоты в трансляторе выполняется единовременная генерация машинного кода для всего исполняемого контекста. Также в режиме JIT-трансляции не выполняются проверки типов данных и не используется модуль логики исполнения.

В качестве стека целочисленных данных кодогенератор использует аппаратный стек x86 с вершиной в регистре, а в качестве вещественного стека используется аппаратный стек сопроцессора x87.

Часть командного набора (включая команды, зарегистрированные клиентом) не транслируются напрямую за отсутствием семантики. Вместо этого, для них генерируется последовательность инструкций, вызывающая виртуальный процессор для исполнения конкретной команды. Ввиду того, что JIT-компилированный код использует аппаратный стек и регистр флагов, команды, исполняемые таким образом, имеют доступ только к виртуальным регистрам и секции данных.

Уровень абстракции

Задачей уровня абстракции является предоставление платформе различной ОС-зависимой функциональности — например, возможности нестандартной работы с памятью. В настоящий момент с помощью уровня абстракции реализованы следующие функции:

- отладочная печать
- выделение памяти с необходимыми правами доступа
- безопасная обработка исключительных ситуаций (сигналов)

Необходимо заметить, что наличие любой из этих функций не является обязательным для запуска платформы. Так, подсистема работы с памятью используется исключительно в модуле JIT-компиляции с целью выделения участка памяти, обладающего правом исполнения. Она реализована для POSIX с использованием функций mmap()/munmap() и для Win32 с использованием функции VirtualAlloc()/VirtualFree().

Для обеспечения безопасности исполнения осуществляется перехват системных ошибочных ситуаций и их обработка с помощью стандартного механизма исключений C++. Это особенно важно в режиме JIT-компиляции, так как почти никаких проверок в этом режиме не производится. Перехват же исключений (в частности, ошибки сегментации) позволяет в случае неполадок корректно завершить выполнение приложения или передать управление интерпретатору для безопасного исполнения проблемного кода.

Данная подсистема реализована для POSIX с использованием интерфейса управления сигналами (signal.h); для остальных систем используется более обобщённая функция signal(), определённая в стандарте ANSI C. Возможна альтернативная реализация этой подсистемы для Win32 с использованием механизма структурной обработки исключений (SEH), но она не была осуществлена ввиду нехватки времени и учебного характера работы.