отобрать сотрудников, участвующих в тестовой эксплуатации, составить список требуемых электронных учебных материалов.

Существует 3 подхода к созданию учебных материалов:

- Приобретение готовых учебных материалов, если удастся их найти.
- Профессиональная разработка учебных материалов. Качественный, но медленный и затратный путь.
- Разработка учебных материалов силами преподавателей. Практически неизбежны кустарный вид, неоднородное качество и проблемы с авторскими правами.

До окончания тестовой эксплуатации рекомендуется обходиться базовым функционалом СДО Moodle и избегать любых программных модификаций или установки нестандартных модулей.

Важно проработать жизненный цикл учетных записей пользователей СДО Moodle: регистрация и удаление, назначение и отписка с учебных курсов, передача итогов обучения, переводы, отчисления, академические отпуска и др. Если в организации уже внедрены информационные системы со списками слушателей и сотрудников, то интеграция с ними является единственным корректным путем.

К работе над этим и последующими этапами следует привлечь специалиста по СДО Moodle и технологиям LAMP, на которых она построена. Совместно с ним проектируется интеграция, требуемые модификации и дополнительные модули, развертывание СДО для тестовой и промышленной эксплуатации.

В. В. Яковлев, Д. В.,Хачко, А. Г. Кушниренко, М. А. Ройтберг
Москва, Пущино, НИИСИ РАН

Проект: Кумир `http://niisi.ru/kumir`, `http://gitorious.org/kumir2`

Ускорение выполнения Кумир-программ с помощью LLVM

Аннотация

Система [1] программирования Кумир, использующая Школьный Алгоритмический Язык [2], предназначена для первичного обучения программированию. Поэтому в версиях Кумир 1.x не было уделено внимание скорости выполнения программ. В версии Кумир 2.0 [3] удалось

на порядок увеличить быстродействие интерпретатора Кумир, в частности, за счет использования более простого внутреннего представления интерпретируемой программы. В версии Кумир 2.1 для Linux были предприняты усилия для дальнейшего увеличения скорости выполнение программ. Это было достигнуто, используя промежуточное представление программы в виде биткода LLVM [4].

Общие сведения

Существует особый класс задач, которые могут решаться с помощью Кумир, но при этом предъявляют высокие требования к производительности: задачи школьных олимпиад по программированию. Правильные решения олимпиадных задач, как правило, имеют алгоритмическую сложность не выше $O(N^k)$, где N — размер исходных данных; k зависит от конкретной задачи. Возможно правильные, но неэффективные решения имеют экспоненциальную сложность либо полиномиальную сложность степени $k' > k$.

При автоматизированной проверке решений олимпиадных задач, задаются ограничения на время выполнения. На вход программы подаются тестовые входные данные, размер которых достаточен для того, чтобы выявить неэффективные решения. Нормативное время выполнения программ, как правило, подбирается эмпирически в расчете на использование компилируемых (Си, Паскаль) языков программирования. Таким образом, школьники, которые используют интерпретируемые языки программирования, оказываются в невыгодном положении.

Способы генерации машинного кода

Существует два подхода для генерации машинного кода: AOT (*A*head *O*f *T*ime) и JIT (*J*ust *I*n *T*ime). В первом случае генератор кода выполняется ровно один раз во время компиляции программы, либо инсталляции на целевом компьютере; во втором — генерация кода выполняется непосредственно перед выполнением программы. Преимуществом AOT является, как правило (но не всегда), более высокая производительность полученного кода, а преимуществом подхода JIT – компактность выполняемого кода, хранимого в промежуточном представлении, и более высокая переносимость.

При проектировании системы Кумир рассматривалось несколько вариантов существующих систем генерации кода с помощью промежуточного представления.

1. **ECMA-335**[5]**,** более известный как Microsoft Intermediate Language. Помимо Microsoft, используется системами программирования PascalABC.NET и Delphi, которые работают только в ОС Windows. Имеет возможность генерации как AOT так и JIT. Стандарт представления является открытым и хорошо документированным, однако существующие реализации среды выполнения .NET и MONO используют патенты, принадлежащие корпорации Microsoft.
2. **Байткод виртуальной машины Java (JSR-000924)**[6]**,** Используется не только компилятором с языка Java, но и со многими другими языками программирования: Scheme, Clojure, Groovy и т. д. Помимо эталонной реализации Sun/Oracle среды выполнения, существует несколько открытых реализаций: IcedTea, OpenJDK, Apache Harmony, благодаря чему байткод Java выгодно отличается от MSIL в плане переносимости.
3. **Промежуточное представление в виде Си или C++ программы.** Этот способ был апробирован[7] в одной из ранних реализаций системы Кумир 1.x и был одной из первых реализаций в прототипе Кумир 2.x. Были выявлены два существенных (особенно в среде Windows) недостатка: необходимость включения в поставку полного комплекта инструментов для компиляции C/C++, и медленная работа самого компилятора C++. Преимуществом генерации в C/C++, помимо высокой скорости выполнения (используется подход AOT), является полная унификация кода runtime-библиотеки с обычной интерпретируемой версией системы Кумир.
4. **Биткод LLVM[8].** Этот способ является дальнейшим развитием метода использования существующего C/C++ компилятора, при этом из последовательности перевода «Кумир → C/C++ →Промежуточное представление → Машинный код» исключается перевод «C/C++ → Промежуточное представление», который является наиболее медленной стадией трансляции. Полученное промежуточное LLVM-представление может быть как выполнено в режиме JIT-компиляции, так и предварительно скомпилировано в машинный код.

Реализация в системе Кумир

В системе Кумир версии 2.1 основным методом выполнения программ в GUI-режиме является интерпретатор. Этот же интерпретатор доступен в виде инструмента командной строки (`kumir2-run`) в поставках для всех поддерживаемых ОС. Реализация компилятора в системе Кумир 2.x является модульной и состоит из двух обособленный частей: анализатор программ (frontend) и генератор выполняемого кода (backend).

Анализатор программ выполняет разбор текста программы и строит дерево ее разбора. Генератор кода обходит это дерево и формирует выполняемую программу. Версия 2.0 включала в себя только один генератор, который строил байткод для входящего в поставку интерпретатора. Версия Кумир 2.1 (пока только для в варианте для Linux) включает в поставку еще один генератор – генератор биткода LLVM. Входящий в поставку инструмент `kumir2-llvm` создает либо машинный код, либо, с помощью задания ключей командной строки, исходный биткод LLVM. Для генерации машинного кода, полученный LLVM-биткод передается стороннему инструменту `clang`, который является интерфейсом для запуска цепочки процессов: `llc`→`as`→`ld`.

Особенностью генератора кода в Кумир является полная эквивалентность выполнения сгенерированных программ соответствующим программам, выполняемым с помощью интерпретатора. Это достигается за счет двух факторов:

1. Стандартная библиотека языка Кумир (точнее, ее расширение, поддерживающее операции ввода-вывода и некоторые другие функциональности) реализована на C++ без использования сторонних библиотек. Один и тот же исходный код стандартной библиотеки компилируется либо как часть интерпретатора, либо, с помощью компилятора CLang, – в биткод LLVM, который затем включается в генерируемый код программ.

2. Все операции языка Кумир, которые требуют контроля во время выполнения (численное переполнение, обращение к элементам массивов, передача и массивов в виде аргументов и т. д.), реализованы единообразно как для интерпретатора, так и для LLVM-биткода – в виде функций C++, предварительно скомпилированных CLang, а затем включаемых в генерируемую программу.

Такая реализация, с одной стороны, почти даром обеспечивает полную идентичность выполнения программ по сравнению с эталонным интерпретатором. С другой стороны, вызов внешних, по отношению к LLVM-программе, C++-функций существенно снижает производительность: такие функции нельзя реализовать как встроенные (inline) фрагменты кода, поэтому программа выполняет много лишних вызовов через стек.

Для увеличения производительности, в последующих реализациях код на C++ постепенно будет заменен на генерацию соответствующих LLVM-фрагментов там, где это представляется рациональным, а для контроля эквивалентности реализации будет использоваться тестирование в процессе непрерывной интеграции.

Сравнение производительности

Для сравнения производительности на различных языках программирования были реализованы три тестовые программы:

1. Алгоритм Флойда-Уоршелла поиска кратчайших путей между парами вершин. Этот алгоритм имеет сложность $O(n^3)$. В качестве входных данных использовалась целочисленная матрица 256×256 весов, предварительно сформированная случайным образом; измерялось время выполнения 100 вызовов алгоритма.

2. Поиск числа перестановок в целочисленном массиве, используя метод сортировки слиянием. Этот тест, помимо оценки производительности работы с массивами, является тестом на эффективность многократного рекурсивного вызова функций. В качестве входных данных использовался предварительно сформированный набор случайных чисел; измерялось время выполнения 100 вызовов алгоритма.

3. Вычисление коэффициентов ряда Фурье для полинома. Вещественные коэффициенты вычисляются в цикле и нигде не сохраняются для того, чтобы можно было оценить производительность только самих вычислений. Поскольку оптимизатор компилятора Си расценивал этот код как не обязательный к выполнению, то вычислялась простая сумма вычисленных коэффициентов. Рассчитывался ряд из 10^7 пар коэффициентов разложения квадратичного полинома $x^2 + 5x + 3$; интегралы вычислялись с точностью 0.1.

При сравнении производительности использовались следующие комбинации языков, компиляторов и их опций:

1. FreePascal с включенной оптимизацией по времени выполнения (опция `-OG`).
2. FreePascal с контролем целочисленного переполнения (опция `-Co`), переполнения стека (опция `-Ct`) и выхода за границы массива (опция `-Cr`). В отличии от языка Кумир, FreePascal не имеет возможности контроля наличия значения у переменной или элемента массива.
3. Язык ANSI C90, компилятор CLang 3.3, без оптимизации (опция `-O0`). Использование языка Си с отключенной оптимизацией позволяет оценить, что именно выполняется процессором, и тем самым – делать теоретическую оценку скорости выполнения отдельных инструкций программы.
4. Язык ANSI C90, компилятор GCC 4.8.1, общеупотребительная оптимизация (опция -O2).
5. Язык Java в реализации от Oracle (версия 1.7.0) с автоматическим использованием JIT-компилятора (опция `-Xmixed`). Эта реализация интересна с точки зрения оценки производительности JIT по сравнению с AOT.
6. Язык Python в реализации CPython версии 2.7.5. С его помощью оценивалась производительность хорошо оптимизированного, но тем не менее – интерпретатора, со свойственными всем интерпретаторам недостатками.
7. Язык Кумир в реализации, описанной выше.

Производились измерения самих алгоритмов, исключая операции ввода-вывода. Результаты измерений приведены в таблице 1.

Результаты

1. Реализован компилятор с языка программирования Кумир в машинный код. Корректность работы этого компилятора подтверждена соответствием результатов выполнения полученных с его помощью программ, соответствующим эталонам, входящим в корпус тестов [9], который предназначен для интерпретатора Кумир.

Язык (опции)	Алг. Флойда		Сорт. слиянием		Коэфф. Фурье	
	Вр., мс	×C, раз	Вр., мс	×C, раз	Вр., мс	×C, раз
Pascal (`fpc -OG`)	10 540	0.96	8 210	0.69	44 880	0.91
Pascal (`fpc -Cr -Co -Ct`)	26 200	2.38	13 280	1.12	45 500	0.93
ANSI C (`clang -O0`)	11 030	1.00	11 889	1.00	49 067	1.00
ANSI C (`gcc -O2`)	2 402	0.22	7 830	0.66	48 257	0.98
Java (`java -Xmixed`)	3 798	0.34	6 878	0.58	394 218	8.03
Python (`python2.7`)	439 955	39.89	435 906	36.66	479 834	9.78
Kumir (`kumir2-llvm`)	186 362	16.90	175 447	14.76	142 493	2.90

Таблица 1: Результаты сравнения производительности различных языков программирования. Для каждой реализации приведено абсолютное время работы тестовых программ на ПК с процессором Xeon E3-1240@3.4GHz, и относительное значение, показывающее во сколько раз реализация медленнее по сравнению с не оптимизированной ANSI C-реализацией.

2. Производительность Кумир-программ, создаваемых с помощью нового компилятора, все еще уступает производительности программ, написанных на языке Паскаль. Тем не менее, этого уровня уже достаточно для успешного использования языка программирования Кумир в школьных олимпиадах муниципального уровня для учеников основной школы – наиболее массовых олимпиад, в которых могут принимать участие ученики, использующие Кумир.
   Относительно низкая производительность обусловлена особенностями текущей реализации, в которой существуют резервы для ее дальнейшего улучшения с 3–17-кратного замедления по сравнению с неоптимизированным Си до уровня 2–5-кратного.

3. Компилятор основан на инфраструктуре LLVM, поэтому на данный момент доступен только для Unix-подобных систем. Использование в среде Windows пока невозможно ввиду отсутствия (по состоянию на декабрь 2013 г., версия LLVM 3.3) работоспособного инструмента для связывания (линковки) полученного с помощью LLVM кода со стандартными библиотеками операционной системы. Недостающий инструмент в настоящее время находится в стадии активной разработки [10].

Литература

[1] `http://gitorious.org/kumir2`

[2] *Информатика: 7–9 кл.: Учеб. для общеобразоват. учр.* А. Г. Кушниренко, Г. В. Лебедев, Я. Н. Зайдельман. М.: Дрофа, 2003. 335 с.

[3] А. Г. Кушниренко, М. А. Ройтберг, Д. В. Хачко, В. В. Яковлев *Кумир 2.0: комплилятор и среда выполнения.* VIII Конференция «Свободное программное обеспечение в высшей школе», М.: Альт Линукс, 2013.

[4] "Introduction to the LLVM Compiler System" Chris Lattner Plenary Talk, ACAT 2008: Advanced Computing and Analysis Techniques in Physics Research, Erice, Sicily, Italy, Nov. 2008.

[5] `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-335.pdf`

[6] `http://docs.oracle.com/javase/specs/jvms/se7/html/index.html`

[7] М. А. Ройтберг, В. В. Яковлев *Конвертор КУМИР – C++: поддержка перехода от учебных к профессиональным системам программирования.* Третья конференция «Свободное программное обеспечение в высшей школе». М.: AltLinux, 2008.

[8] "LLVA: A Low-level Virtual Instruction Set Architecture", Vikram Adve, Chris Lattner, Michael Brukman, Anand Shukla, and Brian Gaeke. Proceedings of the 36th annual ACM/IEEE international symposium on Microarchitecture (MICRO-36), San Diego, California, Dec. 2003.

[9] А. В. Карпов, Е. А. Святушенко, Н. М. Субоч, М. А. Ройтберг *Методы тестирования в разработке системы обучени япрограммированию КуМир.* IV конференция «Свободное программное обеспечение в высшей школе». М.: AltLinux, 2009.

[10] `http://lld.llvm.org/`