Популярные языки программирования с позиций системного программирования

В.В. Косенко

Аннотация. Современное системное программирование не столько низкоуровневое программирование, сколько разработка больших многоуровневых проектов. Языки, применяемые в этой области сейчас, отвечая на эту тенденцию, ставят продуктивность разработки выше эффективности на этапе исполнения. В статье рассматриваются примеры, и предлагается концепция нового соответственно сбалансированного языка.

Ключевые слова: системное программирование, языки программирования, парадигмы программирования, интерпретация, just-in-time - компиляция.

Введение

Системное программирование имеет своей целью создание некой среды или системы, обеспечивающей все необходимое для работы прикладных программ – инструментов в решении конкретных пользовательских задач. Первые системы выполняли лишь простейшее абстрагирование оборудования, напрямую взаимодействуя с интерфейсами устройств. Все средства системного программирования тогда должны были предоставлять в первую очередь примитивы низкоуровневого программирования. Со временем от систем потребовалось больше функций (в том числе из-за виртуализации). В их составе стали появляться блоки, работающие с аппаратурой не напрямую, а через уже существующие и менее высокоуровневые абстракции. Когда объемы исходного кода таких многоуровневых систем стали исчисляться миллионами строк, а команды разработчиков – сотнями человек, на первый план вышла проблема продуктивности разработки, связанная с несоразмерностью: сложности алгоритмов, сложности соответствующего им кода. Сложность можно понимать как перегруженность отделенными по смыслу понятиями.

Рассмотрим алгоритм прохода по списку. Сам алгоритм состоит из понятий списка и операции получения следующего элемента. Код, реализующий алгоритм, на Perl – это цикл foreach по @-списку (два понятия), а код на C++ – цикл for по итератору контейнера списка (четыре понятия). Ясно, что продуктивность Perl-программиста и C++-программиста будут разными, и определяется это возможностями языков. Нестатическая типизация, развитое метапрограммирование, декларативность, исключения, автоматическое управление памятью, встроенные строки и хеш-таблицы повышают продуктивность. Раньше эти возможности использовались только в прикладном программировании, но теперь стали востребованы и в системном программировании.

Примерно 40% официальных пакетов современного дистрибутива debian linux lenny реализовано с помощью изначально прикладных Perl, Python, Java, Haskell и Lisp, и это несмотря на вытекающее из этого менее эффективное, по сравнению с Си, использование вычислительных ресурсов. Нужно признать, продуктивность разработки часто оказывается предпочтительнее эффективности исполнения. Но остаются области системного программиро-

вания, для которых ограничения времени и памяти остаются ключевыми. Речь идет о низкоуровневых элементах операционных систем (ОС), компиляторах, серверов, баз данных, мультимедийных кодеков.

В данной работе рассмотрим популярные языки программирования с позиций системного программирования, останавливаясь на вопросах реализации, связанных с обеспечением, как продуктивности разработки, так и эффективности исполнения. Отрывочное представление о современном состоянии этого вопроса известно, но сколько-нибудь полного анализа с классификацией на текущий момент нет. Этот пробел и призвана восполнить данная статья.

Постановка задачи

Начнем с определения способа оценки эффективности исполнения. Объективно непонятно, какие задачи выбрать для решения на разных языках и проведения сравнений, но со временем программистское сообщество выработало набор тестов, к результатам которых стали относиться как в определенной степени достоверным. Далее мы будем ссылаться на данные одного из таких тестов – The Computer Language Benchmark Game [1] для Intel Q6600, усредненные[1] и вычисленные относительно C++, чьи показатели будем считать за эталон. Показателей всего два: процессорное время и объем оперативной памяти. Данные теста периодически обновляются. Здесь приведем сводку, актуальную на сентябрь 2011 г., а для некоторых языков исключенных из тестирования в апреле того же года – соответственно на апрель, выделяя названия последних курсивом. Переходим к примерам.

Среди языков, отстающих от эталона эффективности более чем в полтора раза (по памяти или по времени) можно выделить одну большую группу – языки, исполняемые на виртуальных машинах. Технологически виртуальные машины – это или интерпретаторы, или just-in-time (JIT) компиляторы. Дифференцировать способы реализации помогают разработчики соответствующих виртуальных машин (они

Табл. 1. Данные для интерпретируемых языков

X/C++	Время исполнения	Используемая память
Lua	25x	2x
CPython	28x	3.4x
Ruby 1.9	38x	35x
Perl	48x	2.2x
Mozart/Oz	52x	28x

либо указывают эту информацию на страницах официального сайта проекта, либо уточняют ее на форумах или по почте).

При интерпретации базовым конструкциям языка ставят в соответствие подпрограммы, вызываемые по ходу разбора исходного кода или сгенерированного из него байт-кода. Как видно из Табл. 1, такой подход требует значительных вычислительных ресурсов.

При JIT-компиляции в цикл интерпретации встраивается компилятор с кэшем, устраняющий необходимость повторной полноценной интерпретации блоков кода, выполняемых несколько раз (тела циклов, функции). Рассмотрим описание алгоритма в LuaJIT [2]:

1. Интерпретируем исходный код. Распознаем начало трассы – той самой повторяющейся части программы (включая переходы, проверку типов, вызовы методов и так далее). Если трасса есть в кэше, то перескакиваем через нее и выполняем соответствующий ей машинный код, иначе переход к 2.

2. Продолжаем интерпретацию, записывая поток выполнения и состояние окружения, соответствующие текущей трассе, производя профилирование. Переходим к 3.

3. Компилируем трассу, используя результаты профилирования, в машинный код и сохраняем его в кэше интерпретатора. Возвращаемся к 1.

Ключевым этапом алгоритма является этап определения границ трасс, тех динамических фрагментов программы, которым можно сопоставить скомпилированный статический аналог. В остальном JIT разбивается на классическую интерпретацию и компиляцию. Второй шаг алгоритма может показаться «лишним», но именно за счет профилирования JIT компилятор может генерировать иногда даже более эффективный, чем статический компилятор, код.

[1] Для времени – медиана, для памяти – среднее арифметическое.

Табл.2. Данные для JIT-языков

X/C++	Время исполнения	Используемая память
Java 7 -server	1x	10.2x
LuaJIT	2x	1.5x
C# Mono	2x	3.8x
F# Mono	3x	6x
JavaScript V8	3x	15.8x
Racket	4x	10x
Erlang HiPE	8x	6.8x
Python PyPy	19x	16x

Как показывают данные из Табл. 2, такой подход значительно эффективнее интерпретации[2].

Видно, что JIT, несмотря на свою технологичность, проигрывает на этапе исполнения C++: если максимально сократить разрыв по времени (Java), то возрастает разрыв по памяти, в среднем по двум показателям получается отставание в 1,5-2 раза (LuaJIT). Реализация JIT с учетом необходимости комбинировать динамический и статический код может оказаться существенно сложнее реализации обычного компилятора, что в свою очередь ограничит, например, переносимость языка на другие платформы.

К сожалению, способа оценки продуктивности программирования на конкретном языке нет. Очевидно, что есть связь между количеством инструментов языка и продуктивностью, которую он может обеспечить. В общем случае можно считать, что это прямая зависимость. Известно, что от прикладных языков, которые и реализуются в виде виртуальных машин, нельзя требовать (и обычно не требуется) высокой эффективности исполнения. Поэтому их развитие никогда не было стеснено технически, и с точки зрения широты выразительных возможностей, а значит, продуктивности, все они сейчас находятся примерно на одном, самом высоком среди других языков, уровне. Но интерес для современного системного программирования вызывают не сами эти возможности как таковые, а то, как их реализовать, когда необходимо эффективное исполнение.

Рассмотрим статически компилируемые языки, очень близкие по эффективности к C++. Начнем с анализа пары языков C/C++. Си – это инструмент реализации Unix и пример баланса между сложностью механизмов языка и их востребованностью. Арифметика указателей, разименование, типы для символьных данных и чисел с плавающей точкой до их появления в Си, не были нужны. BLISS и BCPL были востребованы и без типов (точнее говоря только с целочисленными типами). Язык C++ построен на основе Си из множества неортогональных возможностей. Разноплановость, нововведений, реализованных крайне эффективно, сделали язык очень популярным. Но данная разработка относится к 70-80-м годам и поэтому не учитывает результатов, связанных с виртуальными машинами, получившими широкое распространение сравнительно недавно. По сравнению с ними язык C++ слишком перегружен и всегда менее эффективен как инструмент разработки.

В дальнейшем развитии компиляторов, по нашему мнению, выделяется три направления:

1. Собственно развитие C++.

2. Стремление пересмотреть и/или упростить в целом удачную концепцию C++.

3. Стремление создать нечто новое, кардинально отличающееся от C++.

По первому направлению разрабатываются и обособленные от стандарта диалекты (Tick-C [3], Cyclone [4], OpenMP и др.), и новые версии компилятора (Clang [5]), и сам стандарт продолжает развиваться (C++09 [6]). Этот процесс нацелен на добавление новых возможностей с максимальным сохранением обратной совместимости. На деле язык становится все сложнее, как семантически, так и синтаксически, при этом уровень абстракций остается тем же, и методология программирования не меняется.

Второе направление представлено целым рядом языков. Рассмотрим некоторые из них. Язык Objective-C (ObjC) представляет собой альтернативный по сравнению с C++ взгляд на ООП, с другим набором базовых механизмов, где вызов методов – это пересылка сообщений, а не вызов процедур. Данная концепция более согласована, но передача сообщения оказываются в 1,5 раза медленнее процедур C++ на этапе исполнения. Выбранный нами источник

[2] Популярные ныне Clojure и Scala не имеют собственной реализации и основываются на CLR/Java-машинах, их показатели общей картины не меняют, поэтому в таблице их нет.

[1] не содержит данных тестов на этот счет, поэтому цифры мы получили сами, выполнив простейший тест. Программа выполняет заданное аргументом число вызовов метода. Суммарное время всех вызовов для версии программы на ObjC (y2) делится на время версии на C++ (y1). В Табл. 3 приведена конфигурация машины, результаты – в Табл. 4, исходный код и способ компиляции – в Табл. 5.

Язык D [7] считается переработанным C++ с учетом опыта применения, к примеру, без множественного наследования и со строками, ассоциативными массивами и сборкой мусора в базовом языке. В языке есть практически все, но он перегружен понятиями еще больше C++, но у него нет единого стандарта и на практике он почти не используется.

Язык Go [8], появившийся совсем недавно, является компилируемым и включает в себя упрощенную виртуальную машину (среду времени исполнения) для работы утиной типизации, многопоточности, исключений, сборки мусора и динамических проверок «безопасности» указателей. Компилируемость обеспечивает эффективное исполнение, а виртуальная машина – продуктивность разработки. В реальности Go медленнее C++ в 3 раза [1]. Авторы объясняют это незрелостью своего компилято-

Табл.3. Конфигурация

OS	Debian Lenny
CPU	Intel(R) Xeon(R) CPU E5507 @ 2.27GHz
Instruction set	64-bit
Cache	4 MB
Flags	fpu de tsc msr pae cx8 sep cmov pat clflush mmx fxsr sse sse2
RAM (MemFree)	2.4 GB

Табл.4. Результаты

argv[1]	y1 (C++)	y2 (ObjC)	y2/y1
10^6	0.021	0.03	1.428
10^7	0.178	0.309	1.735
10^8	1.824	2.859	1.567

Табл.5. Тестовые задачи

x.cpp

```
#include <stdlib.h>

class Foo {
public:
  int bar(float x){
    if (x > 0.5)
      return 1;
    else
      return 0;
  }
};

int main(int argc,char** argv) {
  int x = atoi(argv[1]);
  Foo* obj = new Foo();

  while(x--)
    obj->bar(rand());
  return 0;
}



//g++ x.cpp –O3 –o y1
```

x.m

```
#import <objc/Object.h>

@interface Foo : Object {}
-(int) bar: (float) x;
@end

@implementation Foo
-(int) bar: (float) x {
  if (x > 0.5)
    return 1;
  else
    return 0;
}
@end

int main(int argc,char** argv) {
  int x = atoi(argv[1]);
  Foo *obj = [Foo new];

  while(x--)
    [obj bar: rand()];
  return 0;
}

//gcc -x objective-c x.m -lobjc -O3 -o y2
```

ра и тем, что C++ слишком полагается на оптимизированные библиотеки вроде boost, которые не имеют прямого отношения к языку как таковому. Как бы то ни было даже минимальная версия виртуальной машины требует дополнительных ресурсов. Есть и другая проблема. Обработка прерываний, лежащая в основе ОС, не может выполняться на языке с фоновым процессом, блокирующим время от времени основной. В Go такой процесс – виртуальная машина. Как вывод, мало сократить присутствие виртуальной машины, ее нужно реализовать определенным образом или вовсе от нее отказаться. Продолжать эксплуатировать концепцию классических Си-подобных компиляторов тоже нельзя.

Итак, на первый план выходит третье направление развития компилируемых языков и работы по внедрению в системное программирование нестандартных подходов. Сейчас наблюдается повышенный интерес к декларативной парадигме, а именно функциональным и логическим языкам. До недавнего времени эти языки нельзя было использовать повсеместно в системном программировании. Императивность компьютеров требует от декларативных языков не только перевода декларативных описаний в набор императивных инструкций, но и возможности использовать такие инструкции (явно или нет) на уровне программирования. Например, все классические системные алгоритмы основаны на чисто императивной концепции изменяемого состояния. Эмуляция изменяемого состояния часто приводит к нетривиальным дополнительным построениям: в функциональном Haskell-это «нечистые» функции и монады [9], в логическом Prolog – продукции с побочными эффектами и отсечения [10]. Позитивным моментом является тот факт, что декларативная парадигма не исчерпывается функциональной и логической моделями исполнения. Чуть более одного-двух лет назад декларативность на базе автоматического доказательства теорем была реализована в ATS [11] на уровне эффективности C++ [1]. Богатая система типов, отключаемая автоматическая сборка мусора, шаблоны, исключения и интерфейс с Си делают язык крайне перспективным. Но отсутствие полной документации и нестандартная модель исполнения, в настоящее время, ограничивают сферу применения этого языка академическими проектами.

Раньше, до появления Си системщиками активно использовался Fortran, рассматриваемый теперь только в приложении к счетным задачам. В последние несколько лет начался процесс становления современной версии этого языка. Стандарты Fortran 2003 и Fortran 2008 включают возможность совместного использования кода Fortran и Си, поддержку ООП, механизмы динамического выделение памяти и новые способы описания параллельных вычислений. Учитывая способности компиляторов Fortran к оптимизации, можно решить, что обновления стандарта «вернут» язык в системное программирование. Но если посмотреть на список нововведений внимательнее, можно заметить, что в язык добавили только те конструкции, которые в том или ином виде уже есть в наследниках Си.

Основной путь развития системного программирования – это собственно развитие C++, так как остальные направления, включая JIT, реальной альтернативы предложить не могут. Развитие C++, как уже отмечалось, сопряжено с добавлением новых возможностей, которые усложняют язык и «раздувают» компилятор. Если заменить текущие возможности C++ на подобные, но более согласованные и простые, и заложить в язык способ добавления новых конструкций без переписывания компилятора, то эта проблема в целом будет решена. В качестве соответствующего решения нами была сформулирована концепция языка 0xFB[3], в основе которой лежит исходная версия Си с арифметикой, if-ом, while-ом, структурами и указателями, дополненная анонимными функциями и замыканиями для поддержки ООП как в JavaScript[14]. Это компилируемый императивный язык со статической типизацией и элементами функционального программирования, который вполне может эффективно исполняться, и в котором есть современные возможности, повышающие продуктивность разработки.

[3] Концепция исходит из идеи языка 7fa2.L [12], далее доведенной до реализации в виде компилятора CSeL [13]. На данный момент концепция 0xFB уже далеко отошла от этих работ.

Расширяемость в языке реализуется за счет типизированных quasi-quotation макросов со специальным типом для нескомпилированных выражений. Пример кода:

```
def (for n body)(int n, _ body) = {
var int i = 0;
while (i < n) (body; i += 1)
};
for 100 ( print 'Hello,world!' )
```

Параметр n компилируется при поиске подходящего макроса и далее используется его значение. Параметр body является нескомпилированным выражением. Соответствующий узел синтаксического дерева компилируется только внутри макрорасширения в заданном месте. Данный подход можно считать упрощением Nemerle [15, 16] без TypeBuilder'ов и механизмов управления гигиеничностью. Последнее реализуется в 0xFB выбором скобок – фигурных, создающих область видимости, или круглых, осуществляющих лишь группировку. В настоящее время обсуждается выбор модели памяти для 0xFB, но front-end его компилятора и большая часть back-end'а уже реализованы (на C#). Результаты будут опубликованы сразу после окончательной доводки первой полной версии компилятора.

Заключение

В статье были рассмотрены популярные языки программирования с позиций системного программирования, их выразительные возможности и способы реализации трансляторов, предложена классификация. Ключевыми параметрами сравнения языков были эффективность исполнения (на основе тестов) и продуктивность разработки (на основе общих рассуждений). Было показано, что языки, реализованные «поверх» виртуальных машин, не могут быть использованы в системном программировании из-за недостаточной эффективности, а статически компилируемые языки идут путем простого усложнения C/C++, так как другие варианты не могут по ряду причин быть альтернативой.

Ввиду невозможности бесконечного наращивания C/C++, актуальна разработка новых языков системного программирования. В статье сформулированы соответствующие требования и предложено возможное решение в виде концепции 0xFB, сочетающей в себе сильные стороны виртуальных машин и статической компиляции.

Литература

1. Benchmark Game. http://shootout.alioth.debian.org.
2. Pall M. LuaJIT roadmap. http://lua-users.org/lists/lua-l/2008-02/msg00051.html
3. Poletto M. Language and compiler support for dynamic code generation. Ph.D. thesis, MIT, June 1999.
4. Grossman D., Hicks M., Jim T., Morrisett G. Cyclone: A Type-Safe Dialect of C // C/C++ User's Journal, January 2005.
5. Lattner C. LLVM 2.0 and Beyond! // Google Tech Talk, Mountain View, CA, July 2007.
6. Stroustrup B. Trends and future of C++: Evolving a systems language by performance // Leganes, Madrid, March 18th and 28th, 2011.
7. Alexandrescu A. The D Programming Language // Addison-Wesley Professional, June 12, 2010.
8. The Go Programming Language Specification. http://golang.org/ref/spec
9. Stewart D., Jansse S. Design and Implementation of XMonad. A Tiling Window Manager // Haskell Workshop, 2007.
10. Paakki J. Prolog in practical compiler writing // The Computer Journal, Volume 34 Issue 1, Feb. 1991.
11. Hongwei Xi. The ATS Programming Language. ATS / Anairiats User's Guide // Working Draft of October 22, 2010.
12. Бахтерев М.О. Язык программирования 7fa2.L // презентация к выступлению на III международной конференции "Информационно-математические технологии в экономике, технике, образовании" / Екатеринбург, УГТУ-УПИ, 2008.
13. Косенко В.В. Разработка компилятора расширяемого языка системного программирования / Диссертация на степень магистра наук // Екатеринбург, УрФУ, 2011.
14. Diaz D., Harmers R. Pro JavaScript Design Patterns // Apress, 2007.
15. Чистяков В.Ю. Язык Nemerle, часть 5 // RSDN Magazine #2-2011.
16. Чистяков В.Ю. Макросы Nemerle, часть 4 // RSDN Magazine #1-2009.

Косенко Виталий Владимирович. Аспирант Уральского федерального университета им. Б.Н. Ельцина. Окончил магистратуру Уральского федерального университета им. Б.Н.Ельцина в 2011году. Автор двух печатных работ. Область научных интересов: языки программирования, компиляция. E-mail: vitally.kosenko@gmail.com