# Использование языков и сред управляемого исполнения для системного программирования

Д. С. Ушаков
dennis.ushakov@gmail.com

С. И. Салищев
sergey.i.salishev@gmail.com

В данной статье рассматриваются проблемы использования языков и сред управляемого исполнения для решения типичных задач системного программирования, таких как разработка виртуальных машин, операционных систем и драйверов аппаратных устройств. Формулируются требования к технологии системного программирования и изучаются примеры использования в качестве такой технологии Java и ECMA C#/CLI.

## Введение

Системное программное обеспечение — это операционная система, модуль операционной системы драйвер аппаратного устройства или разделяемая компонента; таким образом, от его работы могут одновременно зависеть многие прикладные приложения [4, 15].

Цена ошибки в системном программном обеспечении крайне высока. Ошибка может привести к нарушению работы всех использующих его приложений. Тестирование системного программного обеспечения также затруднено, поскольку его исполнение зависит от недетерминированной последовательности событий, а количество всех возможных вариантов чрезвычайно велико. Системное

программное обеспечение требует эффективного доступа к низкоуровневым свойствам аппаратного обеспечения и гарантий производительности. Все это накладывает требования на технологии разработки системного программного обеспечения, принципиально отличающиеся от требований к разработке прикладного ПО.

За последние 20 лет в области разработки языков программирования произошли большие изменения. Современные языки прикладного программирования предоставляют более выразительную систему типов, приложения, создаваемые с их помощью, более устойчивы к ошибкам, однако для нужд системного программирования по-прежнему используются C и C++ [42].

Увеличение разнообразия аппаратного обеспечения ведет к пропорциональному увеличению объемов системного программного обеспечения и стоимости ошибок, что делает старые модели разработки неприемлемыми с экономической точки зрения [48]. Требуется технология, которая позволила бы существенно сократить стоимость владения кодом, как с точки зрения затрат ресурсов на разработку и сопровождение, так и «тяжести» ошибок, оставшихся в коде.

Время разработки и стоимость владения кодом для языков управляемого исполнения, таких как Java, C#, Ada и Erlang, существенно меньше по сравнению с языками C и C++. Так, согласно исследованию Evans Data Corporation, разработка на C/C++ может быть на 50% более трудоемкой, чем разработка на Java [13]. Продуктивность разработки при использовании языков управляемого исполнения может быть в 4–5 раз больше по сравнению с C/C++ [47]. При этом количество критических дефектов в программах на Java может быть в 3 и более раз меньше, чем в программах с аналогичной функциональностью на C/C++ [2,39]. Эти выводы подтверждаются также в работах [21, 22, 31, 45].

Все это показывает, что языки управляемого исполнения являются заманчивыми кандидатами для использования в системном программировании. В данный момент существуют два популярных стандарта: Java и ECMA C#/CLI [1], из них ECMA C#/CLI обладает большими возможностями низкоуровневого программирования.

Целью данной статьи является обобщение опыта авторов в области разработки системного программного обеспечения с использованием языков управляемого исполнения [1, 9, 25]:

[1] Language popularity. http://www.langpop.com

1) выделение отличительных особенностей системного программирования;

2) формулирование требований к технологии системного программирования;

3) сравнение Java и ECMA C#/CLI как претендентов на роль такой технологии;

4) рассмотрение примеров использования Java и C#/CLI.

## 1. Отличительные особенности системного программирования

Можно выделить следующие особенности, характерные для системного программного обеспечения.

**Особая модель исполнения программ.** Для системного ПО характерна малая гранулярность исполнения. Обычно его задачей является обработка событий аппаратной части, запросов прикладных приложений и других системных событий. При этом сложность обработки одного события невелика, а количество данных, образующих состояние программы и сохраняемых между запусками, значительно превышает количество данных, обрабатываемых во время одного запуска [42].

**Доступ к аппаратной части.** Системное программирование подразумевает эффективный низкоуровневый доступ к аппаратной части платформы.

**Большое значение производительности.** В системном ПО несколько лишних операций могут стоить потери производительности в разы. В отличие от прикладного приложения, где единственной «жертвой» низкой производительности является само приложение, системное ПО не имеет права на такие потери, потому что от него зависит скорость выполнения других программ. Следствием этого является необходимость тонкой подстройки критических участков кода вплоть до уровня инструкций целевой платформы. Также в связи с низкой гранулярностью исполнения использование различных «оберток» и «прослоек» является неприемлемым.

**Управление памятью.** При трехкратном запасе по памяти современные алгоритмы сборки мусора способны на равных конкурировать с «ручным» распределением памяти, а при пятикратном и существенно превосходить его за счет более эффективного использования аппаратных механизмов кэширования и адресации памяти [28]. Алгоритмы сборки мусора реального времени обеспечивают

гарантированно низкие потери производительности [17]. Но в некоторых случаях использование сборки мусора невозможно, например внутри планировщика задач операционной системы, менеджера памяти или обработчика аппаратных прерываний. Кроме того, увеличение размера рабочего множества памяти может негативно повлиять на эффективность кэширования и катастрофически ухудшить производительность подобных приложений.

**Ограниченный доступ к библиотекам.** Системное ПО зачастую выполняется в окружении, где нет возможности использовать стандартные библиотеки и интерфейсы. Следовательно, технология программирования должна обеспечивать гибкость и выразительность, достаточные для создания необходимого инструментария.

## 2. Требования к технологии системного программирования

Под технологией системного программирования будем понимать языки программирования, системные библиотеки, стандарты представления промежуточного и бинарного кода и необходимый инструментарий разработки.

Технология, основанная на компилируемых языках, является, очевидно, достаточной для нужд системного программирования. Разработчики CLI in C#/CLI проектов считают технологию на основе ECMA C#/CLI также потенциально достаточной для этих целей [30, 33]. Разработчики проекта Java in Java отмечают недостаточность потенциала технологии Java для этих целей и предлагают различные расширения языка Java и инструментария разработки [10, 12, 40, 46].

Системное программное обеспечение не может полагаться на наличие обычной среды исполнения, оно предназначено для создания этой среды. К системному программному обеспечению предъявляются два основных требования по качеству: безопасность и эффективность. В отличие от прикладного программного обеспечения, которое может довольствоваться приемлемым качеством, системное программное обеспечение должно обеспечивать наилучшее значение обоих параметров при сохранении приемлемой стоимости разработки.

Для минимизации усилий разработчиков, затрачиваемых на обеспечение безопасности, требуется максимально автоматизиро-

вать процесс поиска ошибок и, по возможности, полностью исключить наиболее часто встречающиеся классы ошибок, что приводит к повышению уровня абстракции языка. Так, например, поскольку проблема статического анализа совмещения указателей является алгоритмически неразрешимой [34], то единственным практическим решением, гарантирующим безопасность работы с указателями, является повышение уровня абстракции языка, исключающее произвольные операции арифметики указателей. Аналогично является алгоритмически неразрешимой проблема статического анализа отсутствия тупиков для модели параллелизма на основе нитей и блокировок [14]. Для гарантирования отсутствия проблем такого рода необходимо использование более высокого уровня абстракции параллелизма, как, например, это сделано в языках Erlang и JOcaml.

Но, кроме того, во многих случаях требуется оптимизация «вручную» и «здесь и сейчас», особенно в условиях недостаточного качества компилятора, что, наоборот, приводит к понижению уровня абстракции языка.

Язык системного программирования должен быть независимым от среды исполнения и допускать эффективный доступ к низкоуровневым свойствам аппаратного обеспечения [36]. В случае использования языков управляемого исполнения отказаться от среды исполнения практически невозможно, поэтому инструментарий должен обеспечивать возможность порождения бинарного кода аппаратной платформы для эффективной реализации предусмотренных стандартом интерфейсов взаимодействия с окружением. Также он должен поддерживать и кросскомпиляцию.

Технология системного программирования должна обеспечивать возможность простой и эффективной интеграции с внешними библиотеками и интерфейсами, поэтому она должна поддерживать типы данных языка C, поскольку этот язык является индустриальным стандартом описания интерфейсов. При этом преобразование типов, приводящее к копированию данных или изменению семантики значений, обычно является неприемлемым.

Современные прикладные языки создавались исключительно для программирования прикладных, а не системных приложений. Однако многие свойства современных языков могут быть крайне полезны для системного программирования, но многие факторы, повлиявшие на проектирование старых языков, больше не являются существенными.

Безопасные типы полностью определяют все допустимые операции со своими экземплярами [41]. В C/C++ отсутствуют безопасные типы, поскольку есть возможность небезопасного приведения типов и произвольного доступа к памяти. Причина этого двояка. С одной стороны, для некоторых задач требуется прямой доступ к памяти и аппаратному обеспечению. С другой стороны, языки разрабатывались достаточно давно и не были рассчитаны на использование современных оптимизирующих компиляторов, поэтому использование безопасных типов данных приводило к неэффективности. В связи с развитием компиляторов последняя причина больше не действует, наоборот, компиляторы могут использовать дополнительную информацию о типах для эффективной оптимизации [5, 18, 19, 26].

Одна из ключевых особенностей современных языков прикладного программирования — это представление об управляемом исполнении кода, обеспечивающее отсутствие невосстановимых ошибок в контексте этого кода. Таким образом любое ошибочное состояние приложения может быть обработано и восстановлено в безошибочное. Это свойство распространяется только на ошибки, связанные с поведением управляемого кода, и не относится к ошибкам неуправляемого кода, операционной системы, внешних модулей, ресурсов и аппаратного обеспечения. Во многих случаях ошибки исчерпания памяти и переполнения стека вызовов, локализованные в управляемом коде, также могут быть восстановлены [11, 29]. Это разительно отличается от неуправляемого исполнения кода, когда «затирание» переменных на стеке или выход за границы выделенного блока памяти приводит к невосстановимым последствиям. Использование управляемого кода обеспечивает существенное сокращение ресурсов на тестирование и отладку и уменьшает количество и серьезность оставшихся ошибок. Использование управляемого кода сохраняет актуальность и для системного программного обеспечения, поскольку большая часть кода системы может быть написана без использования низкоуровневых операций [30, 32, 47].

Основным недостатком управляемого кода является строгая высокоуровневая семантика, препятствующая низкоуровневому программированию. Для решения этой проблемы в прикладном программировании используются интерфейсы вызова внешних функций (Foreign Function Interface), например JNI. Это приводит к большим накладным расходам, связанным с невозможностью межпроцедурной оптимизации. Межпроцедурная оптимизация во мно-

гих случаях улучшает производительность хорошо структурированного кода в несколько раз [16]. Невозможность такой оптимизации приводит к существенному увеличению гранулярности небезопасного кода для сохранения характеристик производительности. Для системного программирования этот фактор может сделать применение управляемого кода невозможным.

Требуется эффективный механизм иерархического понижения уровня безопасности исполнения кода, сочетающий возможность межпроцедурной оптимизации и доступа к небезопасным типам, тонкой оптимизации на уровне бинарного кода, эффективного включения вставок на ассемблере аппаратной платформы, доступа к интерфейсам операционной системы, внешним библиотекам.

Раздельная компиляция модулей, применяемая в C, негативно сказывается на эффективности кода. Для повышения эффективности требуется сквозная межпроцедурная оптимизация. Кроме того, желательно сохранить удобства частичной компиляции (быстрота и инкрементальность) [35, 38, 43].

В системном программировании часто возникает задача выделения временной памяти в контексте вызова. При этом часто нет возможности положиться на эффективность сборщика мусора или же выделение памяти в куче невозможно, например, при сборке мусора, вызове библиотечных методов, написанных на C/C++, внутри примитивов синхронизации и других участках кода, где требуется детерминированный порядок исполнения. Для эффективного решения этой задачи требуется поддержка со стороны языка и среды исполнения.

Как уже было отмечено, одним из преимуществ традиционных языков программирования является проверенный опытом инструментарий разработки, который позволяет производить раскрутку компиляторов. Зрелость технологии определяется наличием такого инструментария и возможности произвести самосборку инструментария, необходимого для программирования с использованием этой технологии.

Подведем итоги. Необходимыми требованиями к технологии системного программирования являются:

- использование управляемого кода;
- иерархическое понижение уровня безопасности кода;
- соответствующий контроль типов на всех уровнях безопасности;

- эффективная поддержка типов C при взаимодействии с внешними интерфейсами;
- сквозная оптимизация, в том числе низкоуровневых вставок;
- типизированное выделение временной памяти в контексте вызова функции;
- наличие инструментария с возможностью порождения бинарного кода аппаратной платформы и кросскомпиляции.

## 3. Сравнение языков Java и CLI

Стандарт ECMA CLI [43] был принят в 2006 году, спустя 11 лет после официального выпуска Java компанией Sun Microsystems. При разработке данного стандарта были учтены многие ошибки, допущенные при создании Java, в частности крайне важные для системного программирования. Как уже было замечено, одним из свойств языка для системного программирования является обеспечение доступа к низкоуровневым свойствам аппаратного обеспечения или эффективная интеграция с традиционными низкоуровневыми средствами разработки, такими как C и ассемблер.

Одной из серьёзных ошибок в программировании на C является неявное приведение знакового типа к беззнаковому. В Java, в целях избежания подобных ошибок, было принято радикальное решение отказаться от беззнаковых типов. Это приводит к существенным ограничениям в задачах, связанных с системным программированием. Так, например, возникают сложности при реализации алгоритмов сжатия и шифрования, сетевых протоколов и файловых форматов. Также возникают сложности при необходимости передать параметры беззнаковых типов в бинарный код (например, в библиотеку, написанную на C) или получить их обратно. В CLI был учтён этот негативный опыт, и беззнаковые типы заняли своё законное место.

Для взаимодействия с библиотеками, написанными на C/C++ или ассемблере, в Java был предложен специальный интерфейс — Java Native Interface (JNI). Он крайне сложен, требует описания взаимодействия как на стороне Java, так и на обратной, и накладывает существенные ограничения на использование типов и имён. Таким образом, для того чтобы переиспользовать старый код, необходимо написать специальный адаптирующий слой на C/C++. Технология Platform Invoke (P/Invoke), являющаяся общепринятым расширением CLI, предоставляет ту же возможность, однако пе-

редача параметров и результатов описывается в атрибутах метаданных, что значительно упрощает задачу программиста: ему не нужно писать дополнительный адаптирующий слой. Следует, однако, заметить, что использование этих технологий крайне негативно сказывается на производительности приложения.

В языке Java весь код является безопасным: проверка типов является обязательной, отсутствуют указатели. В CLI программисту разрешается отключить проверку типов и другие возможности системы безопасности при помощи ключевого слова unsafe.

Небезопасный код представляет собой аналог кода на языке C и сочетает гибкость, кроссплатформенность, близость к аппаратной платформе и возможность межпроцедурной оптимизации с управляемым кодом. При этом программисту даётся возможность обращаться к низкоуровневым свойствам системы: использовать указатели, в частности указатели на функции, обращаться к бинарным библиотекам. В силу реализации данный подход оказывается более быстрым, нежели Platform Invoke.

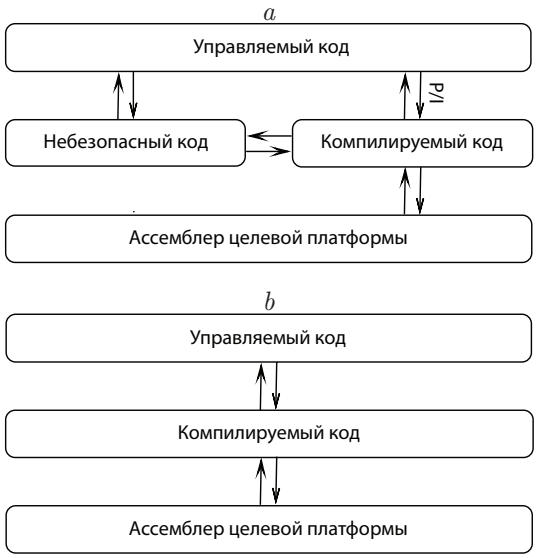
Использование небезопасного кода, P/Invoke и JNI требует крайней осторожности и при некорректном использовании может привести к «падению» самой виртуальной машины. Небезопасный код является наиболее удобным способом взаимодействия с низкоуровневыми компонентами (см. рисунок).

Одним из требований к технологии системного программирования является возможность производить типизированное выделение временной памяти в контексте вызова функции. CLI предоставляет такую возможность — для этого вводятся пользовательские типы-значения. В базовом стандарте Java такой возможности нет, размещение в контексте вызова функции может произойти только в результате оптимизации JIT-компилятором и поэтому не представляет практической ценности для задач системного программирования.

Возможность типизированного выделения временной памяти в контексте вызова функции реализована в Real-Time Java[2].

Инструментарий разработчиков, как Java, так и ECMA CLI, достаточно хорошо развит, но оба стандарта не предусматривают никакого механизма трансляции управляемого кода в код целевой платформы, что осложняет раскрутку виртуальной машины, написанной на этих платформах. Однако две наиболее используемые ре-

[2]Real-Time Java. http://java.sun.com/javase/technologies/realtime/index.jsp




Доступ к низкоуровневым средствам в ECMA CLI и Java:
$a$ — CLI; $b$ — Java.

ализации ECMA CLI (Microsoft .Net и Mono[3]) предоставляют AOT-компиляцию для процессоров x86. Другие реализации, такие как Singularity и SharpOS, реализуют свои компиляторы из IL в код целевой платформы.

## 4. Опыт использования Java

Одной из главных особенностей всех проектов Java-in-Java являются модульное проектирование и упрощение применения современных техник программирования, таких как шаблоны проектирования и agile-методы разработки [6, 7, 27, 37].

Система Jikes RVM [10] (ранее известная как Jalapeno) разрабатывалась в IBM с конца 1997 года и является доказательством возможности написания быстрой виртуальной машины на Java. Единственные части Jikes RVM, написанные на C, — это небольшой загрузчик и «обёртка» интерфейса взаимодействия с операционной

[3]Mono: http://www.mono-project.com/

системой. В Jikes RVM входит «агрессивный» JIT-компилятор с целевыми платформами PowerPC и IA32. В феврале 2002 года, Jikes RVM достигла 95% производительности IBM 1.3.0 DK JVM на Linux/IA32 в тесте SPECjvm98, наглядно продемонстрировав возможность написания высокопроизводительной виртуальной машины Java-на-Java. В то время как в оптимизирующем компиляторе используется вся полнота языка Java, части ядра и оригинальный сборщик мусора активно использовали статический код, что отражает прошлое некоторых авторов как программистов на C, и относительную новизну языка Java в то время, когда был написан код. Система управления памятью MMTk [8] была написана позднее авторами с глубокой верой в оптимизирующий компилятор. Однако из соображений корректности (таких, как невозможность вызова `new()` во время работы сборщика) использовались статическая аллокация и нетипизированные указатели, что является существенным искажением семантики Java и неудобно для программирования. Для раскрутки Jikes RVM использует бинарную сериализацию [3].

Система Squawk [20] — это виртуальная машина Java для встроенных устройств, реализующая профиль CLDC (Connected Limited Device Configuration) 1.1. Разработчики Squawk избежали большинства проблем, связанных с использованием Java-в-Java, благодаря применению подмножества языка Java, представимого при помощи языка C. Такой подход позволяет использовать компилятор C для сборки Squawk. Приложение исполняется при помощи интерпретатора либо может быть предварительно транслировано в код на C. Реализация JIT-компилятора не предусмотрена архитектурой этой виртуальной машины, что не позволяет расширить Squawk для эффективной реализации Java SE (Standart Edition).

На базе Squawk был реализован экспериментальный драйвер для Solaris [49]. Авторам удалось добиться производительности, в худшем случае всего в 3.8 раза уступающей производительности драйвера, написанного на чистом C. Однако необходимо заметить, что устройством, для которого был написан драйвер, был RAM-диск, что позволило избежать многих проблем работы с реальной аппаратной частью.

Существует также ряд других JVM, написанных на языке Java. Среди них Joeq [46] (разработка компиляторов), JNode [40] (операционные системы) и Rivet [12] (тестирование). Во время работы над проектами Moxie [9] и Jikes RVM были выявлены недостатки

языка Java для системного программирования, затрудняющие портирование кода C, эффективное взаимодействие с внешними библиотеками и написание низкоуровневого кода.

Проект Moxie [9], разрабатывавшийся при непосредственном участии авторов, развивает идеи Jikes RVM и борется с её основными недостатками. Именно при работе над этим проектом были сформулированы основные требования к технологии системного программирования, а также выявлены слабые стороны Java как кандидата на эту роль.

В 2006 году компанией Sun Microsystems была начата разработка экспериментальной виртуальной машины Maxine[4] с использованием языка управляемого исполнения Java. Основными задачами данного проекта являются:

- исследование принципов разработки архитектуры виртуальных машин;
- повышение настраиваемости и гибкости виртуальной машины;
- развитие технологий построения виртуальных машин.

Эти задачи, как и пути их решения, во многом повторяют цели и задачи, поставленные разработчиками проектов Jikes RVM и Moxie. Например, в Maxine для раскрутки также используется бинарная сериализация.

Подводя итоги отметим, что основной проблемой языка Java является отсутствие средств низкоуровневого программирования. На уровне среды управляемого исполнения эти средства могут быть достаточно эффективно поддержаны при помощи идиом, например vmmagic [9, 24]. Эффективность использования идиом для низкоуровневого программирования доказана применением vmmagic для сквозной оптимизации реализации барьеров и выделения памяти в DRL VM [25], написанной на C/C++. На уровне языка Java применение идиом приводит к «раздуванию» и запутыванию кода. Также расширения нарушают совместимость со спецификацией JVM, сужают область применения кода и затрудняют тестирование и отладку.

[4]Maxine Project. https://maxine.dev.java.net/

## 5. Опыт использования ECMA CLI

Проект Singularity стартовал в 2003 году в исследовательском подразделении корпорации Microsoft для создания высоконадёжной операционной системы (ОС), в которой микроядро, драйверы устройств и приложения написаны в рамках парадигмы управляемого кода.

Отличительной особенностью данной ОС является использование идеологии программно-изолированных процессов (SIP, похожих на «легкие» процессы языка Erlang), общение между которыми происходит исключительно посредством сообщений. В отличие от традиционных ОС, защита таких процессов в Singularity производится не путем организации аппаратно-защищенных адресных пространств, а с помощью использования типобезопасного подмножества промежуточного языка (MSIL) и его верификации перед компиляцией в бинарный код процессора [23].

Ядро системы Singularity предоставляет основные абстракции программно-изолированных процессов, каналов и их контрактов, а также программ и манифестов. Ядро играет ключевую роль в распределении системных ресурсов между программами и сокрытии сложностей внутреннего устройства аппаратного обеспечения. Каждому процессу оно предоставляет чистое пространство для исполнения с нитями, памятью и доступом к другим программам посредством каналов. Подобно аналогичным проектам Cedar [44] и Spin, Singularity использует сборку мусора и статическую проверку типов. Примерно 90% ядра Singularity написано на Sing#, специальном диалекте C#. Хотя бóльшая часть ядра написана на типизированном языке Sing#, значительная часть ядра разработана на небезопасном варианте этого языка. Наиболее важной частью, написанной в небезопасном окружении, является сборщик мусора, который составляет примерно 48% небезопасного кода. Другие источники небезопасного Sing# — это подсистемы управления памятью и система ввода/вывода. В Singularity есть небольшие участки ассемблерного кода в тех же местах, в которых они есть и в ядрах, написанных на C/C++, например в местах переключения контекста и в векторах прерываний. Отладчик ядра и код низкоуровневой инициализации ядра написаны на C++, что составляет примерно 6% кода ядра Singularity.

Для «раскрутки» Singularity используется компилятор Bartok, который преобразует код сборок CLI в код целевой платформы

(x86). Также компилятор генерирует заголовочные файлы, необходимые для связывания кода на Sing# с кодом целевой платформы. Затем компилируются 16-битный и 32-битный загрузчики, после чего происходит связывание кода на C/C++ с объектными файлами, собранными Bartok.

При разработке Singularity стояла задача сделать код операционной системы, драйверов и исполняемых программ верифицируемым и не накладывалось никаких ограничений на язык реализации (примерно 10% кода написано на C/C++ и ассемблере целевой машины), а задачей аналогичного проекта SharpOS [33] является написание операционной системы только с использованием языка C#. В отличие от Singularity, SharpOS является открытой операционной системой, распространяемой под лицензией GPL v.3 с исключениями для библиотеки классов. Изначально рассматривались два пути решения проблемы раскрутки: реализовать среду исполнения на традиционном языке исполнения или же расширить C# низкоуровневыми средствами. Однако небезопасные возможности C# были признаны достаточными для написания кода ядра операционной системы. Для раскрутки применяется статическая компиляция IL в код целевой платформы с введением дополнительного уровня абстракции в коде ядра и AOT-компилятора, который позволяет легко сменить целевую платформу.

Существуют также и другие реализации операционных систем на ECMA CLI: Cosmos (C# Open Source Managed Operating System) и Ensemble OS (распределённая операционная система), однако о них представлено крайне мало информации в отличие от Singularity.

Проект IKVM.NET[5] — это открытая реализация виртуальной машины Java на ECMA CLI. Для построения виртуальной машины производится статическая рекомпиляция стандартной библиотеки классов из OpenJDK и части классов GNU Classpath в промежуточное представление CIL, Java программы рекомпилируются на лету. Несмотря на двойную компиляцию (Java Byte Code -> CIL -> код платформы), на стандартных тестах IKVM.NET показывает производительность, сравнимую с Sun Java 1.6, а иногда и превосходит оригинал. Однако данный подход при построении виртуальной машины нарушает спецификацию JVM и лицензионное соглашение OpenJDK.

[5] http://www.ikvm.net

В ходе дипломной работы [1] авторами на базе IKVM.NET была разработана совместимая с GPL2 и спецификацией Java Virtual Machine среда Java Runtime Environment. При постановке задачи рассматривалась необходимость объединить две популярные среды исполняемого управления в одной. В качестве технологии разработки был избран ECMA CLI. В ходе данной работы был переписан системный загрузчик классов и улучшено взаимодействие виртуальной машины со стандартной библиотекой классов. Трудностей, возникавших при работе над проектом Moxie, в частности в отладке основных компонент виртуальной машины, удалось избежать благодаря выбору в качестве технологии разработки ECMA CLI.

## Заключение

Технологии системного программирования на основе языков и сред управляемого исполнения доказали свою состоятельность на примере множества экспериментальных проектов и некоторых промышленных решений.

С одной стороны, все концептуальные и технические проблемы подхода уже, кажется, решены в рамках проекта Singularity [30] на базе ECMA C#/CLI. С другой стороны, адаптация технологии тормозится закрытостью исследовательской лицензии Microsoft и отсутствием открытых эффективных реализаций стандартов. Поэтому широкого применения технологии можно ожидать только после адаптации самой Microsoft в рамках коммерческих продуктов.

На счету Java-технологии большая распространенность, наличие нескольких качественных открытых реализаций[6], успешный опыт практического применения Real-Time Java [22, 48]. В рамках проекта Jikes RVM ведется активная разработка безопасных средств низкоуровневого программирования [25]. Но пока открытым остается вопрос поддержки системного программирования на уровне языка.

Возможным сценарием развития технологий системного программирования видится симбиоз обоих стандартов в рамках единого инструментария, что представляется наиболее предпочтительным и наименее затратным, учитывая их внутреннее. Другой вариант — разработка нового языка или диалекта для системного программирования в рамках технологии Java.

[6]Open JDK project page, http://openjdk.java.net/

## Список литературы


[1] *Ушаков Д. С.* Разработка JRE на ECMA CLI. Дипломная работа. СПбГУ, 2008. http://endal.ath.cx/_media/mos_thesis.doc

[2] *Шипилев А. Д.* Эволюционные алгоритмы в задачах метаоптимизации Java JIT компиляторов. Дипломная работа. СПбГУИТМО, 2009. http://people.apache.org/~shade/Shipilev_Evolutionary_-Algorithms_In_Metaoptimization_of_Java_JIT_Compilers.pdf

[3] *Alpern B., Attanasio C. et al.* The jalapeño virtual machine // IBM Syst. J. 2000. Vol. 39. N 1. P. 211–238.

[4] *Back G., Hsieh W. C.* The KaffeOS Java Runtime System // ACM Trans. Program. Lang. Syst. 2005. Vol. 27. N 4. P. 583–630.

[5] *Bacon D. F., Fink S. J., Grove D.* Space- and Time-Efficient Implementation of the Java Object Model // ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming. London, UK. Springer-Verlag, 2002. P. 111–132.

[6] *Beck K., Beedle M. et al.* Manifesto for Agile Software Development. 2001. http://agilemanifesto.org/

[7] *Blackburn S. M., Boehm H., et al M. C.* Second Moxie Brainstorming Meeting. Jan 2006. http://moxie.sf.net/

[8] *Blackburn S. M., Cheng P., McKinley K. S.* Oil and Water? High Performance Garbage Collection in Java with MMTk // ICSE '04: Proceedings of the 26th International Conference on Software Engineering. Washington, DC, USA. IEEE Computer Society, 2004. P. 137–146.

[9] *Blackburn S. M., Salishev S. I. et al.* The Moxie JVM Experience: Tech. Rep. TR-CS-08-01: Australian National University, Department of Computer Science. Jan 2008.

[10] *Bowen Alpern, Attanasio C. R. et al.* Implementing Jalapeño in Java // OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications. New York, NY, USA. ACM, 1999. P. 314–324.

[11] *Boyland J.* Handling «Out of Memory Errors» // ECOOP 2005 Workshop on Exception Handling in Object-Oriented Systems. Jul 2005.

[12] *Bruening D., Chapin J.* Systematic Testing of Multithreaded Programs: Tech. Rep. MIT-LCS-TM-607. MIT. May 2000.

[13] *Butters A. M.* Total cost of ownership: A comparison of C/C++ and Java: Tech. rep.: Evans Data Corporation. Jun 2007. http://whitepapers.techrepublic.com.com/abstract.aspx?docid=342102

[14] *Cieslak R. A., Variaya P. P.* Undecidability Results for Deterministic Communicating Sequentialprocesses // *Automatic Control, IEEE Transactions on*. Sep. 1990. Vol. 35. P. 1032–1039.

[15] *Czajkowski G., Daynès L., Titzer B.* A Multi-User Virtual Machine // ATEC '03: Proceedings of the Annual Conference on USENIX Annual

Technical Conference. Berkeley, CA, USA. USENIX Association, 2003. P. 7.

[16] *Cooper L. T., Bixby R.* Managing Interprocedural Optimization. 1991.

[17] *David F., Bacon P. C., Rajan V. T.* A Real-Time Garbage Collector with Low Overhead and Consistent Utilization // *SIGPLAN Not.* 2003. Vol. 38, N 1. P. 285–298.

[18] *Dean J., Chambers C. et al.* Vortex: an Optimizing Compiler for Object-Oriented Languages // OOPSLA '96: Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. New York, NY, USA. ACM, 1996. P. 83–100.

[19] *Dean J., Chambers C., Grove D.* Selective Specialization for Object-Oriented Languages // PLDI '95: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation. New York, NY, USA. ACM, 1995. P. 93–102.

[20] *Doug D. S., Cifuentes C.* The Squawk Virtual Machine: Java™ on the Bare Metal // OOPSLA '05: Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. New York, NY, USA. ACM, 2005. P. 150–151.

[21] *Christiansen C. Q.* Java Pays — Positively // *IDC Bulletin #W16212*. 1998. http://www.usdoj.gov/atr/cases/exhibits/1344.pdf

[22] *Edward G., Benowitz A.* Experiences in Adopting Real-Time Java for Flight-Like Software // On The Move to Meaningful Internet Systems 2003: OTM 2003Workshops. 2003. Vol. 2889/2003. P. 490–496.

[23] *Fähndrich M., Aiken M. et al.* Language Support for Fast and Reliable Message-Based Communication in Singularity OS // EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006. New York, NY, USA. ACM, 2006. P. 177–190.

[24] *Flack C., Hosking T., Vitek J.* Idioms in Ovm: Tech. Rep. CSD-TR-03-017: Purdue University Department of Computer Sciences, 2003.

[25] *Frampton D., Blackburn S. M., Salishev S. I.* Demystifying Magic: High-Level Low-Level Programming // VEE 2009: The 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. New York, NY, USA. ACM, 2009.

[26] *Gagnon E. M., Hendren L. J.* SableVM: A Research Framework for the Efficient Execution of Java Bytecode // Java Virtual Machine Research and Technology Symposium. USENIX, 2001. P. 27–40.

[27] *Gamma E., Helm R. et al.* Design Patterns: Elements of Reusable Object-Oriented Software. Boston, MA, USA. Addison-Wesley Longman Publishing Co., Inc., 1995.

[28] *Hertz M.* Quantifying the Performance of Garbage Collection vs. Explicit Memory Management // Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA. ACM Press, 2005. P. 313–326.

[29] How Does the JVM Recover from a java.lang.stackoverflowerror? http://forums-beta.sun.com/thread.jspa?messageID=9752556

[30] *Hunt G. C., Larus J. R.* Singularity: Rethinking the Software Stack // SIGOPS Oper. Syst. Rev. 2007. Vol. 41, N 2. P. 37–49.

[31] *Jessup J.* Java from the Horse's Mouth // *Softw. Dev.* 1999. Vol. 7, N 3. P. 73.

[32] Jikes RVM Project Page. http://jikesrvm.org/

[33] *Lahti W.* SharpOS: We are the Chicken and The Egg (draft). http://www.sharpos.org/redmine/wiki/3/Draft_Publicity_Article

[34] *Landi W.* Undecidability of Static Analysis // ACM Letters on Programming Languages and Systems. 1992. Vol. 1. P. 323–337.

[35] *Lindholm T., Yellin F.* The Java Virtual Machine Specification. Second Edition. 2nd edition. Prentice Hall, 1999.

[36] *McCorkle E. L.* Modern Features for Systems Programming Languages // ACM-SE 44: Proceedings of the 44th Annual Southeast Regional Conference. New York, NY, USA. ACM, 2006. P. 691–697.

[37] *McDirmid S., Flatt M., Hsieh W. C.* Jiazzi: New-Age Components for Old-Fasioned Java // OOPSLA '01: Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. New York, NY, USA. ACM, 2001. P. 211–222.

[38] *Palacz K., Baker J. et al.* Engineering a Common Intermediate Representation for the Ovm Framework // Science of Computer Programming. 2005. Vol. 57(3). P. 357–378.

[39] *Phipps G.* Comparing Observed Bug and Productivity Rates for Java and C // Software — Practice and Experience. 1999. Vol. 29. P. 345–358.

[40] *Prangsma E.* Why Java is Practical for Modern Operating Systems // Libre Software Meeting. 2005. Presentation only. http://www.jnode.org

[41] *Saraswat V.* Java is not Type-Safe. 1997. http://www.citeseer.ist.psu.edu/saraswat97java.html

[42] *Shapiro J.* Programming Language Challenges in Systems Codes: Why Systems Programmers Still Use C, and What to Do about it // PLOS '06: Proceedings of the 3rd Workshop on Programming Languages and Operating Systems. New York, NY, USA. ACM, 2006. P. 9.

[43] Standard Ecma-335: Common Language Infrastructure (CLI). http://www.ecma-international.org/publications/standards/Ecma-335.htm

[44] *Swinehart D. C., Zellweger P. T. et al.* A Structural View of the Cedar Programming Environment // ACM Trans. Program. Lang. Syst. 1986. Vol. 8, N 4. P. 419–490.

[45] *Tyma P.* Why are we Using Java Again? // Commun. ACM. 1998. Vol. 41, N 6. P. 38–42.

[46] *Whaley J.* Joeq: a Virtual Machine and Compiler Infrastructure // IVME '03: Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators. New York, NY, USA. ACM, 2003. P. 58–66.

[47] *Wiger U.* Four-Fold Increase in Productivity and Quality — Industrial Strength Functional Programming in Telecom-Class Products // Workshop on Formal Design of Safety Critical Embedded Systems. May 2001.

[48] *Wood D.* Java Emerges as Solution for Military Software Modernization // VME and Critical Systems. may 2007. http://www.vmecritical.com/articles/id/?2162

[49] *Yamauchi H., Wolczko M.* Writing Solaris Device Drivers in Java // PLOS '06: Proceedings of the 3rd Workshop on Programming Languages and Operating Systems. New York, NY, USA. ACM, 2006. P. 3.