МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ» (НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет Механико-математический факультет
Кафедра Программирования

Направление подготовки Математика и компьютерные науки

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Васько Мария Богдановна
(Фамилия, Имя, Отчество автора)

Тема работы: Эффективная реализация ассемблерных вставок в компиляторе языка высокого уровня

«К защите допущена»
Вр.и.о. зав. кафедрой,
к.ф.-м.н.

Емельянов П.Г. /_____
(фамилия , И., О.) / (подпись, МП)

«…»……20…г.

Научный руководитель
Старший преподаватель
ММФ НГУ

Трепаков И.С. /_____
(фамилия , И., О.) / (подпись, МП)

«…»……20…г.

Дата защиты: «…»……20…г.

Новосибирск, 2026

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Ассемблерные вставки традиционно играют ключевую роль в системном программировании, позволяя интегрировать низкоуровневые инструкции непосредственно в код высокоуровневого языка. Это обеспечивает точный контроль над аппаратными ресурсами, что зачастую невозможно реализовать средствами исходного языка. Поэтому многие компиляторы высокоуровневых языков, такие как *GNU Compiler Collection* [1], *clang* [2], *Microsoft Visual C++ compiler* [3], *D compiler* [4] и *rustc* [5], предоставляют поддержку ассемблерных вставок.

Подобное встраивание блоков ассемблерного кода позволяет программе напрямую взаимодействовать с регистрами процессора, использовать специфические для архитектуры инструкции и реализовывать системные вызовы, что особенно важно в задачах, требующих строгого контроля над производительностью и ресурсами. К числу таких задач относятся: сборка мусора, реализация примитивов синхронизации, переключение контекстов в системах виртуализации (*KVM* [6], *Hyper-V* [7]) и т.д.

Тем не менее, использование ассемблерных вставок связано с рядом серьёзных ограничений. *Во-первых*, они ограничивают переносимость программ: код, написанный для одной архитектуры, зачастую невозможно без изменений использовать на другой. *Во-вторых*, ассемблерные вставки усложняют процесс разработки и сопровождения программного обеспечения, так как требуют от программиста глубокого знания архитектурных особенностей и понимания протокола

взаимодействия ассемблерного и высокоуровневого кода. *В-третьих*, они представляют собой барьер для оптимизирующих компиляторов: ассемблерный код препятствует выполнению многих трансформаций и мешает полноценному анализу программы.

Решение обозначенных выше проблем продолжает оставаться актуальной задачей при реализации ассемблерных вставок в компиляторах как существующих, так и новых высокоуровневых языков для системного программирования. Для того чтобы такие вставки не становились препятствием для оптимизаций, требуется использовать специальные методы интеграции блоков ассемблерного кода в процесс анализа и трансформации программы.

Так, уже существующие направления реализации ассемблерных вставок включают: интеграцию блоков низкоуровневого кода в промежуточное представление [8], совместимое с инфраструктурой оптимизирующего компилятора, построение формальных моделей семантики инструкций и симуляцию ассемблерного кода на уровне высокоуровневого языка [9], [10]. Эти подходы позволяют сохранить семантику исходного кода и одновременно открывают возможности для дальнейшего применения стандартных методов оптимизации и верификации.

В данной работе рассматриваются подходы к реализации и оптимизации ассемблерных вставок в оптимизирующем компиляторе языка высокого уровня. Цель работы — анализ существующих решений, их ограничений и потенциальных направлений оптимизации,

а также разработка экспериментальной реализации поддержки вставок ассемблерного кода на базе статического компилятора многоязыковой виртуальной машины *Huawei* путем использования низкоуровневого внутреннего представления компилятора в качестве языка написания таких фрагментов.

1 Предметная область

1.1 Ассемблерные вставки

Языки программирования могут быть классифицированы по уровням абстракции, отражающим их степень удаленности от аппаратных средств вычислительной системы. Языки низкого уровня, такие как машинный код и ассемблер, обеспечивают непосредственный доступ к ресурсам процессора: регистрам, стеку, арифметико-логическим устройствам и системе адресации памяти.

Машинный код представляет собой двоичную последовательность инструкций, интерпретируемую центральным процессором напрямую. Ассемблер, в свою очередь, является символьным представлением машинных инструкций. Мнемоники, символические имена регистров и метки делают его удобнее для восприятия человеком, но семантически он остаётся эквивалентным машинному коду.

Ассемблерные вставки (англ. *inline assembly*, *inline asm*) представляют собой фрагменты низкоуровневого кода, встроенные в программу, написанную на языке высокого уровня. Интеграция ассемблерного кода в программу позволяет:

а) использовать инструкции, недоступные из-за высокого уровня абстракции языка;

б) напрямую взаимодействовать с аппаратными ресурсами машины;

в) оптимизировать производительность критических участков кода.

С другой стороны, такие вставки нарушают многие свойства высокоуровневой семантики: композиционность, статическую проверку типов, применимость оптимизаций и инварианты модели исполнения.

1.1.1 GNU Inline Assembly

Наиболее распространенный синтаксис ассемблерных вставок — *GNU inline assembly* синтаксис. Он предоставляет компилятору интерфейс, описывающий способ взаимодействия вставки с окружением. Например, какие ресурсы используются вставкой, какие регистры, флаги или области памяти могут быть изменены.

Для данного синтаксиса выделяют *архитекрутные* диалекты: структура интерфейса в общем случае представлена единым образом, но сам ассемблерный код вставки написан под конкретную архитектуру.

Большинство диалектов следуют официальным спецификациям набора команд соответствующих архитектур. Исключением является архитектура *x86*, где помимо стандартного синтаксиса *Intel* традиционно используется специфичный для инструментов GNU диалект *AT&T*. Далее в примерах мы будем рассматривать вставки, написанные на данном диалекте.

Помимо диалектов сам синтаксис *GNU inline assembly* имеет две вариации — базовую (англ. *basic*) и расширенную (англ. *extended*).

Базовая версия (листинг 1) позволяет вставку ассемблерных инструкций в произвольных местах в коде программы.

```
__asm__ [volatile|inline]
    («инструкции ассемблера»
    );
```

Листинг 1 — базовая версия синтаксиса GNU inline assembly

Важным является то, что компилятор не знает о побочных эффектах ассемблерного кода, таких как изменения памяти или регистров. Поэтому все базовые блоки неявно считаются волатильными (англ. *volatile*). Это предположение ограничивает оптимизатор, запрещая удаление или перемещение кода вставки, но не гарантирует сохранность данных в регистрах, которые могут быть изменены внутри блока.

Расширенная вставка (листинг 2) предоставляет интерфейсом между ассемблером и остальной программой. В отличие от базовой версии, являющейся полностью непрозрачной для компилятора, расширенная вставка вводит декларативный контракт, уточняющий допустимое для использования множество регистров и ресурсов.

```
__asm__ [volatile|inline|goto]
    («инструкции и директивы ассемблера»
    : выходные операнды
    : входные операнды
    : изменяемые операнды
    : список меток
    );
```

Листинг 2 — расширенная версия синтаксиса GNU inline assembly

Опишем то, как в общем случае выглядит использование расширенной вставки в синтаксисе *GNU Inline Assembly*.

Текст вставки на ассемблере представляет собой форматированную строку, содержащую инструкции и директивы ассемблера. Эта строка принимает параметры, именуемые *операндами ассемблерной вставки.*

Определение 1. *Операнды ассемблерной вставки — аргументы, связывающие переменные программы с регистрами или ячейками памяти, используемыми внутри ассемблерного кода.*

Для каждого операнда задаются *ограничения.*

Определение 2. *Ограничения (англ. constraints) — правила, определяющие допустимые способы размещения операндов (регистр, память, константа) и семантику взаимодействия компилятора с ними.*

Ограничения декларируют, какие именно ресурсы (например, регистры определенного типа) допустимо использовать для передачи данных. Например, *«r»* — допустимо размещение операнда на регистре, *«m»* — в памяти, *«i»* — в виде константного значения.

Так, выделяют три типа ограниченных операндов, различающихся по семантике: выходные, входные и изменяемые.

Определение 3. *Выходные операнды (англ. outputs) — операнды ассемблерной вставки, содержащие результирующие значения.*

В синтаксисе *GNU inline assembly* ограничения на выходные операнды задаются префиксом *«=»* (перезапись значения) или *«+»* (при чтении и записи), например, *«=r»*. Это указывает на то, что ассемблер

будет записывать данные в этот операнд, который затем будет доступен в качестве возвращаемого значения.

Определение 4. *Входные операнды (англ. inputs) — операнды ассемблерной вставки, которые доступны для чтения ассемблерным кодом.*

Ограничения входных операндов не могут начинаться ни с *«=»*, ни с *«+»*. Но они могут быть представлены цифрами (например, *«0»*), которые обозначают то, что указанный входной операнд должен находиться в том же месте, что и выходной, по соответствующему индексу в списке выходных ограничений.

Компилятор предполагает, что после завершения выполнения ассемблерного кода эти операнды содержат те же значения, что и до выполнения вставки.

С другой стороны, хотя компилятор знает об изменениях переменных и регистров, перечисленных в выходных операндах, встроенный ассемблерный код может изменять не только их.

Определение 5. *Изменяемые операнды (англ. clobbers) — операнды ассемблерной вставки, указывающие символические имена регистров или другие ресурсы (память, флаги), которые могут быть изменены ассемблерным кодом без их явного перечисления во множествах входных и выходных операндов.*

Важным является то, что множество изменяемых регистров не должно каким-либо образом пересекаться с входными или выходными операндами.

В следствие, когда компилятор выбирает регистры для представления входных и выходных операндов, он не использует изменяемые операнды. Это делает изменяемые ресурсы доступными для свободного использования в коде ассемблера.

Рассмотрим интерфейс базовой и расширенной версий вставки на конкретных примерах (листинги 3, 4).

```
__asm__ volatile
   ("movl %%esi, %%eax\n\t"
    "lock cmpxchgl %%edx, (%%ecx)\n\t"
    "movl %%eax, %%ecx"
   );
```

Листинг 3 — базовая версия вставки (архитектура x86-64)

```
/* Объявление переменных int old_val,
int* addr, int new_value, int expected
*/

__asm__ volatile
   ("movl %3, %%eax\n\t"
    "lock cmpxchgl %2, %1\n\t"
    "movl %%eax, %0"
    : "=r" (old_val), "+m" (*addr)
    : "r" (new_value), "r" (expected)
    : "eax", "cc", "memory"
   );
```

Листинг 4 — расширенная версия вставки (архитектура x86-64)

Передача значений в качестве позиционных аргументов осуществляется по номеру операнда с использованием префикса %, после которого следует опциональный модификатор. Нумерация же начинается с 0 и идет непрерывно, объединяя все элементы списков параметров. Каждая строчка, кроме последней, заканчивается символом

переноса строки. Препроцессор[1] осуществляет подстановку ассемблерных операндов на место номера, учитывая модификаторы операндов и присвоенные им номера.

Помимо указания операндов, интерфейс ассемблерных вставок позволяет указывать метки[2] для перехода по ним в C-коде.

Определение 6. *Список меток — список всех меток в коде, к которым может перейти ассемблерный код.*

Использование ___*asm*___ *goto* позволяет ассемблерному коду переходить к одной или нескольким меткам C, указанных в списке. Эта информация является опциональной, поэтому без ключевого слова *goto* во вставке не указывается список меток. Пример использования представлен на листинге 5.

```
int foo(int count) {
    __asm__ goto
        ("dec %0; jb %l[stop]"
        : "+r" (count)
        : /* Без входных операндов*/
        : /* Без изменяемых операндов*/
        : stop
        );
    return count;
    stop:
        return 0;
}
```

Листинг 5 — пример вставки с использованием ключевого слова *goto*. Тело вставки использует метку *stop*, объявленную внутри функции *foo*

[1]Программа, подготавливающая код программы на языке C/C++ к компиляции.

[2]Метка (англ. label) — символьное имя, идентификатор для более удобного указания данных и кода в языках программирования.

Ссылка на метку в коде вставки требует использования префикса *%l* (строчная буква *L*), за которым следует позиция метки (начиная с нуля) в соответствующем списке.

Несмотря на то, что интерфейс расширенной вставки предоставляет большое количество возможностей для написания ассемблерного блока кода, позволяет компилятору корректно размещать значения на регистрах и предотвращает нарушение целостности данных, он имеет ряд существенных недостатков:

а) не описывает семантику инструкций;

б) не определяет влияние на внутреннее состояние программы;

в) не даёт достаточной информации для проведения сложных оптимизаций и применения методов формальной верификации.

1.2 Реализация вставок в современных статических компиляторах

Способы интеграции ассемблерного кода в современных статичексих компиляторах различаются, но их объединяет изолированность вставок от общей модели зависимостей. По этой же причине компилятор вынужден принимать консервативные решения относительно вставок при внутри- и межпроцедурных оптимизациях и анализах.

1.2.1 GNU GCC

Компилятор *GNU Compiler collection* (GCC) [11] использует предоставляемый *GNU Inline Assembly* интерфейс. Как было сказано

ранее, компилятор не анализирует семантику самих инструкций внутри ассемблерной строки. Он полагается исключительно на декларативный контракт (ограничения и изменяемые операнды) при проведении распределения регистров и оптимизаций.

Рассмотрим на примере кода (листинг 4), каким образом ассемблерная вставка обрабатывается компилятором GCC.

На ранних стадиях компиляции программы ассемблерная вставка представляется в виде узла *ASM_EXPR* промежуточного представления *GENERIC* [12], содержащего строку ассемблера и спецификации операндов.

При трансляции в промежуточное представление *GIMPLE* [12] этот узел преобразуется в *GIMPLE_ASM*, интегрируемый в граф потока управления. На этом уровне GCC фиксирует зависимости между ассемблерной вставкой и компонентами промежуточного представления, устанавливает барьеры для оптимизаций.

На стадии понижения до низкоуровневого промежуточного представления *RTL* [11] узел *GIMPLE_ASM* превращается в *RTL*-структуру, содержащую уже низкоуровневые описания регистров и памяти (Листинг 6).

```
  (parallel [
    (set rtx-for-old_val
        (asm_operands/v:SI "movl %3, %%eax\n\t ... movl %%eax, %0" "=r" 0
            [rtx-for-new_value rtx-for-expected rtx-for-*addr]
            [(asm_input:SI "r")
             (asm_input:SI "r")
             (asm_input:SI "m")]))
    (set rtx-for-*addr ...)
    (clobber (mem:BLK (scratch)))
    (clobber (reg:DI ax))
    (clobber (reg:CC flags))
])
```

Листинг 6 — пример упрощенной RTL-структуры для рассматриваемого примера

Структура объединяет конструкцию *asm_operands*, содержащую сам текст ассемблерных команд, список входных и выходных переменных с их ограничениями, а также перечень регистров и областей памяти, которые будут изменены в ходе выполнения. На фазе генерации ассемблерного кода структура генерируется в выходной файл с подставленными регистрами и адресами.

Так, с течением фаз (рисунок 1) компилятор создает форму в промежуточном представлении, содержащую информацию об ассемблерной вставке для генерации кода.

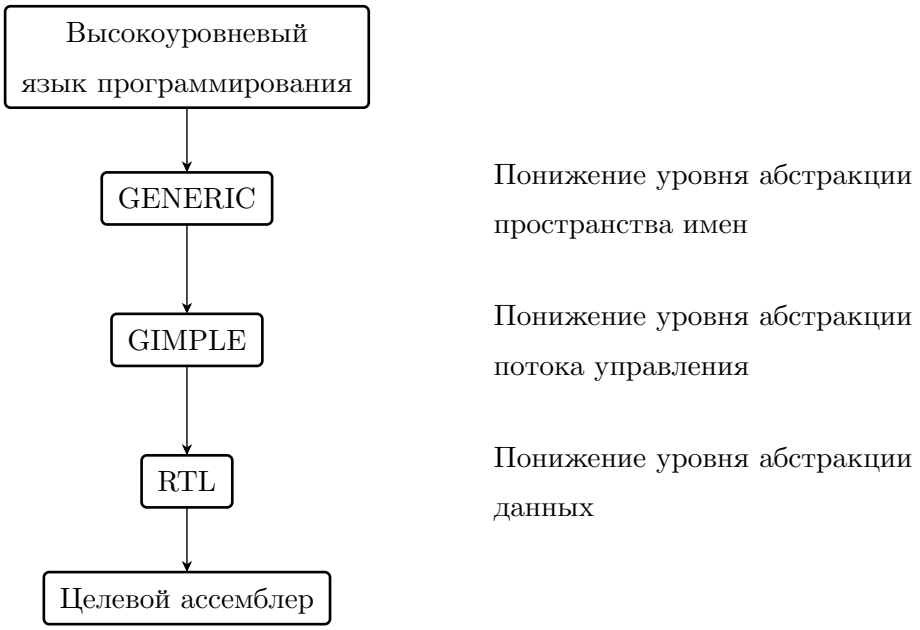


Рисунок 1 — обработка ассемблерной вставки для GCC

Однако *gcc* воспринимает вставку как барьер для оптимизаций. Из-за того, что в GIMPLE/RTL вставка часто помечается как имеющая неизвестные побочные эффекты, компилятор вынужден принимать консервативные решения при проведении многих оптимизаций (например, перенос кода) вокруг неё.

1.2.2 Clang

Рассмотрим так же на примере кода (листинг 4) обработку ассемблерной вставки компилятором *clang* [13], являющимся частью инфраструктуры *LLVM* [14].

Clang проводит [15] парсинг ассемблерной вставки, полагаясь на интерфейс GNU Inline Assembly, и в процессе семантического анализа осуществляет валидацию операндов.

В промежуточном представлении компилятор формирует специальную *bitcode-инструкцию* [16] *call asm(…)*. На этом уровне

представление ассемблерной вставки является специализированной формой *call*-инструкции, содержащей саму строку кода ассемблера, строку с ограничениями, накладываемыми на операнды, и передаваемые аргументы (листинг 7).

```
call i32 asm sideeffect
   "movl $3, %eax\0A\09lock cmpxchgl $2, $1\0A\09movl %eax, $0",
   "=r,=*m,r,r,*m,~{eax},~{cc},~{memory},~{dirflag},~{fpsr},~{flags}"
   (i32* elementtype(i32) %0, i32 %2, i32 %1, i32* elementtype(i32) %0) #1,
   !srcloc !5
```

Листинг 7 — репрезентация примера в промежуточном представлении

Здесь атрибут *sideeffect* запрещает оптимизатору перемещать или удалять код, а строка ограничений явно кодирует типы операндов и список изменяемых регистров через префикс ~.

Это означает, что компилятор воспринимает вставку как внешнюю функцию с неизвестными побочными эффектами, что разрывает граф управления, вынуждает компилятор вставлять барьеры для оптимизации (консервативные решения, например невозможность вынесения кода перед или после вставки) и ограничивает анализы программы.

Для минимизации этих ограничений перед финальной генерацией машинного кода текстовая строка ассемблера транслируется в архитектурно-специфичную инструкцию *INLINEASM*.

```
bb.0 (%ir-block.3):
  liveins: $edx, $esi, $rdi
  INLINEASM &"movl $3, %eax\n\t ... movl %eax, $0" [sideeffect] [mayload] [maystore]
      [regdef:GR32], def $ecx,
      [mem:m],       $rdi,
      [reguse:GR32], $edx,
      [reguse:GR32], $esi,
      [clobber],     implicit-def $eax,
      [clobber],     implicit-def $eflags
  $eax = MOV32rr $ecx
  RET64 $eax
```

Листинг 8 — упрощенное архитектурно-специфичное представление

Информация из строки ограничений теперь представлена в виде явных свойств (*mayload*, *maystore*). Входные, возвращаемые и изменяемые (*reguse*, *regdef*, *clobbers*) ресурсы явно указаны в представлении.

После происходит генерация ассемблерного кода и последующее его понижение в машинные инструкции с использованием ограничений, флагов и опций, полученных в ходе парсинга.

Тем не менее на протяжении всех фаз генерации кода (рисунок 2) компилятор рассматривает инструкцию как «непрозрачный» узел графа потока управления, сохраняющий побочные эффекты, но не имеющий семантики.

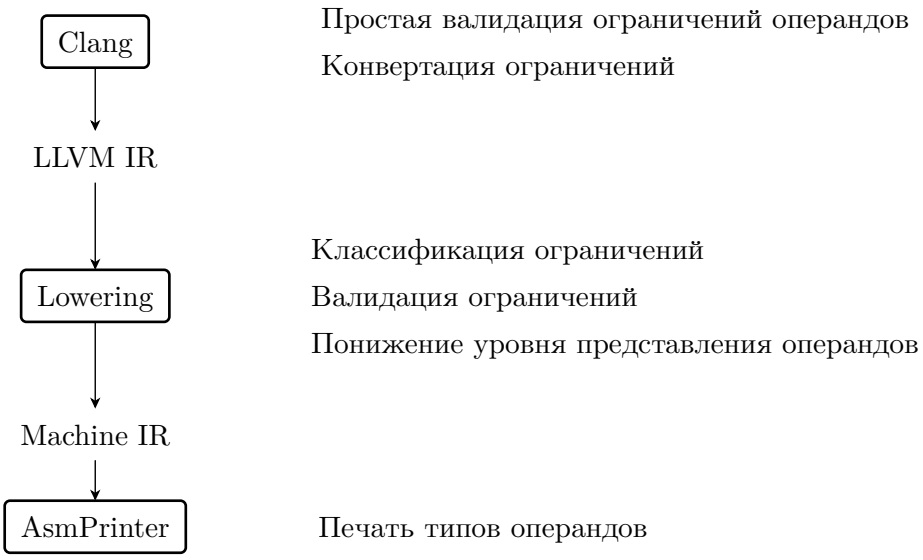


Рисунок 2 — обработка ассемблерной вставки для LLVM

Это создает разрыв между гибкостью промежуточного представления LLVM и жестко зафиксированными вставками.

1.2.3 Rust compiler

Язык *Rust* использует типобезопасную конструкцию *asm!* [5], где операнды объявляются как *in(reg)*, *out(reg)*, *inout(reg)*, есть строгая проверка типов, и компилятор обеспечивает согласованность множеств регистров и размеров данных (листинг 9).

```
/* Объявление переменных mut old_val: i32, addr: *mut i32,
   new_value: i32, expected: i32 */
asm!(
    "movl {3:e}, %eax",
    "lock cmpxchgl {2:e}, ({1})",
    "movl %eax, {0:e}",
    out(reg) old_val,
    in(reg) addr,
    in(reg) new_value,
    in(reg) expected,
    out("eax") _,
    options(att_syntax)
);
```

Листинг 9 — пример макроса *asm!* в Rust

На этапе генерации кода компилятор *rustc* транслирует конструкцию *asm!* в промежуточное представление LLVM аналогично тому, как это делает *clang*, то есть в инструкцию *call asm(…)*. Поэтому семантика ассемблерной вставки так же остаётся непрозрачной для LLVM.

Подобное введение конструкции в язык улучшает интерфейс и расширяет возможности верификации, но не меняет полноту семантики ассемблерной вставки: оптимизатор не получает дополнительной информации.

1.3 Реализация вставок в современных управляемых системах

В отличие от статических компиляторов, где ассемблерный код фиксируется в момент написания, в управляемых средах, таких как виртуальные машины, требуется механизм отложенной генерации. Это

необходимо потому, что прямая вставка ассемблера в высокоуровневый язык, например Java, нарушает бинарную переносимость и безопасность среды исполнения.

1.3.1 HotSpot JVM

В виртуальной машине *Java HotSpot* [17] место текстовых вставок ассемблерного кода динамический компилятор *C2* использует специализированные узлы промежуточного представления *Sea of Nodes*, которые на этапе генерации кода заменяются конкретными машинными инструкциями через файлы описания архитектуры (AD-файлы).

В AD-файле разработчик описывает правила сопоставления и указывает побочные эффекты отдельным полем *effects*, такие как использование конкретных регистров, временных ресурсов и изменение флагов процессора.

Так, на листинге 10 приведен пример определения атомарной операции *CompareAndSwapP* в AD-файле.

```
instruct compareAndSwapP(rRegI res,
                         memory mem_ptr,
                         rax_RegP oldval, rRegP newval,
                         rFlagsReg cr)
%{
  predicate(n->as_LoadStore()->barrier_data() == 0);
  match(Set res (CompareAndSwapP mem_ptr (Binary oldval newval)));
  match(Set res (WeakCompareAndSwapP mem_ptr (Binary oldval newval)));
  effect(KILL cr, KILL oldval);

  format %{ "cmpxchgq $mem_ptr,$newval\t# "
            "If rax == $mem_ptr then store $newval into $mem_ptr\n\t"
            "setcc $res \t# emits sete + movzbl or setzue for APX" %}
  ins_encode %{
    __ lock();
    __ cmpxchgq($newval$$Register, $mem_ptr$$Address);
    __ setcc(Assembler::equal, $res$$Register);
  %}
  ins_pipe( pipe_cmpxchg );
%}
```

Листинг 10 — пример описания вставки compareAndSwapP в Java HotSpot в формате AD-файла для архитектуры x86-64

Данная вставка имеет следующие побочные эффекты: изменения состояния флага *cr* (*Condition Register*) и значения *oldval*. Помимо этого явно указан регистр для этого значения (*rax_RegP*). Тело вставки описывается в секции *ins_encode* с использованием предметно-ориентированного языка для генерации макросов *MacroAssembler*, написанного на языке C++.

Такой подход позволяет в одном месте описать и тело самой вставки, и все ограничения, задаваемые для ее генерации. Поскольку регистры и побочные эффекты зафиксированы в интерфейсе AD-файла,

компилятор заранее видит ограничения вставки и может эффективно вписать её в общий поток кода.

Помимо этого, с точки зрения сборки мусора, которая является неотъемлемой частью управляемых систем, это упрощает генерацию барьеров доступа к памяти, так как планировщик заранее знает, какие ссылки и ресурсы будут изменены.

1.3.2 GraalVM

Компилятор *Graal* [18] реализует интеграцию через объектно-ориентированный интерфейс в низкоуровневом представлении *LIR*. Каждая вставка представляется отдельным классом, расширяющим базовый класс *LIRInstruction*, что позволяет встроить её в общий процесс распределения регистров и планирования инструкций.

Это позволяет описывать требования к регистрам через аннотации, вместо использования внешних файлов описания архитектуры (AD-файлов), как это сделано в HotSpot.

Рассмотрим реализацию атомарной операции *CompareAndSwapOp*, представленной на листинге 11.

```
@Opcode("CAS")
public static final class CompareAndSwapOp extends AMD64LIRInstruction {
    @Def protected AllocatableValue result;
    @Use({COMPOSITE}) protected AMD64AddressValue address;
    @Use protected AllocatableValue cmpValue;
    @Use protected AllocatableValue newValue;
    ...
    @Override
    public void emitCode(CompilationResultBuilder c, AMD64MacroAssembler masm) {
        assert asRegister(cmpValue).equals(AMD64.rax) : cmpValue;
        assert asRegister(result).equals(AMD64.rax)   : result;
        if (c.target.isMP) {
            masm.lock();
        }
        // оператор выбора
        masm.cmpxchgb(address.toAddress(masm), asRegister(newValue));
        ...
        }
    }
}
```

Листинг 11 — исkпользование интерфейса для генерации атомарной операции CompareAndSwapOp в GraalVM для архитектуры x86-64

В данной вставке аннотация *@Use* обозначает аргументы, значения которых читаются, но не изменяются, то есть входные операнды. Результат вставки записывается в регистр, аннотированный *@Def*.

Для описания временных регистров необходимо определить дополнительную переменную в классе и добавить аннотацию *@Temp* *@Alive* указывает компилятору, что значения адреса и ожидаемого операнда должны сохраняться на регистрах на протяжении всей операции и не могут быть перезаписаны. .

В отличие от *HotSpot C2*, в GraalVM нельзя заранее указать в интерфейсе использование конкретных регистров, например регистр

rax во вставке CompareAndSwapOp для архитектуры x86-64 (Amd64). Их приходится назначать прямо в коде при генерации низкоуровневых операций (листинг 12).

```
private Value emitCompareAndSwapHelper(boolean isLogic, LIRKind accessKind,
                                       Value address, Value expectedValue,
                                       Value newValue, BarrierType barrierType) {
      ...
      RegisterValue aRes = rax.asValue(integerAccessKind);
      ...
      emitMove(aRes, reinterpretedExpectedValue);
      emitCompareAndSwapOp(isLogic, integerAccessKind, memKind,
                           aRes, addressValue, allocatableNewValue,
                           barrierType);
```

Листинг 12 — пример генерации CompareAndSwap для архитектуры x86-64. Регистр rax явно передается как аргумент в метод emitCompareAndSwapOp, генерирующий вставку

Тело вставки генерируется с использованием предметно-ориентированного языка для генерации макросов *MacroAssembler*, написанного на языке Java.

1.3.3 Проблематика реализаций

В отличие от полноценной программы на ассемблере, вставка обычно небольшая и выполняет строго локальную функцию. Однако, именно «локальность» приводит к необходимости точной формализации интерфейса взаимодействия.

Наличие корректного интерфейса имеет ключевое значение для оптимизатора, поскольку компилятор обязан гарантировать сохранение

инвариантов для корректного исполнения программы. На практике реализация ассемблерной вставки осложняется следующими факторами:

- вставка зависит от архитектуры, ABI[3], соглашений о вызовах[4] и конкретных расширений процессора;
- интерфейс должен позволять компилятору безопасно интегрировать вставку в оптимизируемый код;
- желательно, чтобы вставка могла быть проанализирована статически либо интегрирована в представление, подходящее для применения формальных методов верификации.

Таким образом, ассемблерные вставки позволяют напрямую использовать инструкции процессора, однако создают значительные сложности для анализа и оптимизации. В современных компиляторах, таких как *gcc, clang, rustc*, ассемблерный код остается изолированным в графе промежуточного представления. Он оборачивается минимальным интерфейсом, задающим набор ограничений, который является для компилятора барьером для оптимизаций.

В отличие от статических систем, в управляемых системах вставка требует согласованности с механизмами среды исполнения. При отложенной генерации кода разработчик обязан гарантировать корректное взаимодействие вставки со сборщиком мусора и средствами управления потоками. В итоге любая неточность в описании

---

[3]Бинарный интерфейс приложения (англ. Application Binary Interface, ABI) — это интерфейс, используемый средой выполнения и операционными системами для выражения двоичных сведений низкого уровня.

[4]Соглашения о вызовах (англ. calling conventions) — часть ABI, описывающая технические особенности вызова подпрограмм.

используемых ресурсов или побочных эффектов может привести к некорректной работе программы.

Это ограничивает возможности компилятора по анализу потоков данных и управлению, создавая мотивацию для исследований в области альтернативных подходов: типизированных многоязыковых систем [10], механизмов бинарной интеграции [19] и моделей с глубокой интеграцией семантики смешанных языков [9].

2 Обзор предшествующих работ

Существует несколько подходов к решению проблем, связанных с существующими реализациями ассемблерных вставок в высокоуровневых языках программирования. В данной работе предлагается к рассмотрению три подхода: (2.1) интеграция вставок в промежуточное представление языка высокого уровня, (2.2) построение смешанной языковой системы, (2.3) интеграция вставок ассемблерного кода посредством построения семантических моделей.

2.1 Интеграция в промежуточное представление.

Ключевая проблема, которая рассматривается в работе [8], — это отсутствие формального способа передачи информации о семантике ассемблерной вставки в оптимизационные фазы компилятора.

Это препятствует как безопасной интеграции блоков ассемблерного кода, так и дальнейшему анализу. Классические подходы (*gcc*, *clang*, *C2*) полагаются на синтаксическое описание вставки, а не на семантику ее инструкций, что делает их неточными.

Для решения этих проблем авторы работы предлагают к использованию подход, основанный на бинарной интеграции (*binary lifting*) — трансляции бинарных инструкций в платформонезависимое промежуточное представление более высокого уровня абстракции.

Ассемблерная вставка дизассемблируется и преобразуется в промежуточное представление *BINSEC IR*[20], в котором явно представлены регистры, флаги, память и зависимости между

ними. Отладочная информация позволяет узнать местоположение ассемблерных вставок в скомпилированном коде.

Следующий важный шаг — преобразование низкоуровневого представления в высокоуровневый код языка C. Чтобы переход был корректным, авторы предлагают к использованию механизм распространения типов, который сопоставляет объекты представления с типами языка C. Типовая информация передаётся как из отладочной информации, полученной при компиляции, так и из ограничений, накладываемых низкоуровневыми операциями. Схема обработки вставки представлена на рисунке 3.

Рисунок 3 — поднятие ассемблерных вставок

Данный подход позволяет получить эквивалентный фрагмент C-кода, отражающий поведение ассемблерной вставки. Так, семантика вставки становится доступной для стандартных оптимизаций компилятора, включая распространение констант, устранение мёртвого кода, анализ живости регистров и оптимизацию доступа к памяти.

Этот процесс не только устраняет необходимость явного объявления интерфейса ассемблерной вставки, но и делает возможным автоматическую проверку её корректности относительно окружающего высокоуровневого кода.

Такое решение позволяет устранить синтаксическую многозначность, достичь архитектурной независимости, проводить глубокий анализ и оптимизации, и делает вставки ассемблерного кода полноценной частью промежуточного представления программы.

2.2 Построение многоязыковой системы.

Определенную сложность в интеграции ассемблерных вставок представляет то, что ассемблерный код не является композиционным[5] – поток управления может быть изменен в произвольной точке кода из-за потенциального использования прямых переходов и доступа к стеку вызовов.

Поэтому, чтобы разрешить эту проблему для высокоуровневых языков, например, функциональных, которые являются композиционными, необходимо ограничить поведение ассемблерных вставок. Это достигается введением *типизированного языка ассемблера*, снабжённого интерфейсами, строго описывающими состояние стека, набор доступных регистров, структуру продолжений[6] и допустимые переходы.

В своей работе [10] авторы рассматривают способ построения многоязыковой системы *FunTAL* для интеграции ассемблерных вставок в функциональные языки.

---

[5]Свойство языка программирования, при котором семантика сложного выражения полностью определяется семантикой его составных частей и правилами их соединения.

[6]Абстрактное представление состояния программы в определённый момент, которое может быть сохранено и использовано для перехода в это состояние.

Реализация такого решения становится возможной благодаря введению логических отношений для доказательства корректности поведения ассемблерных вставок. Построенные отношения позволяют обосновать эквивалентность низкоуровневых вставок и высокоуровневых компонент промежуточного представления.

При этом авторы работы отмечают, что объединение высокоуровневых и низкоуровневых моделей исполнения без рассмотрения семантики ассемблерного кода может привести к нарушению инвариантов программы. Подобная проблема характерна для традиционных встроенных ассемблерных вставок в таких компиляторах, как *gcc* [11] и *clang* [13].

Именно поэтому возникает необходимость определения граничных операторов, которые корректно переводят состояние одной модели исполнения в состояние другой. Граничный оператор обеспечивает корректную передачу управления в высокоуровневую среду, автоматически восстанавливая необходимые абстракции и гарантируя согласованность типов возвращаемых значений.

Важной особенностью предложенного метода является то, что ассемблерный код не только вызывается из функционального языка, но и сам способен инициировать вызов функций исходного языка. Схема взаимодействия представлена на рисунке 4.

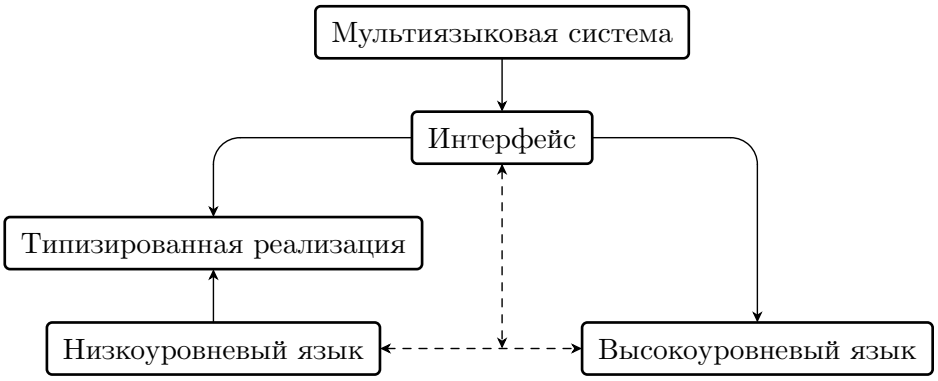


Рисунок 4 — схема взаимодействия моделей исполнения. Пунктирными линиями обозначено взаимодействие двух моделей

В итоге такой механизм интеграции вставок, предоставляющий способ взаимодействия двух уровней абстракций, делает смешанную программу полностью компонуемой: ассемблерные фрагменты могут свободно взаимодействовать с функциональными значениями, включая замыкания и функции высших порядков.

2.3 Интеграция в семантику языка.

В своей работе [9] Андрей Шадрин рассматривает проблематику взаимодействия ассемблерного и высокоуровневого кода с точки зрения среды, в котором оно должно происходить. При отсутствии единой модели исполнения смешанная программа становится семантически неоднозначной: невозможно формально определить, каким образом низкоуровневые изменения стека или регистров отобразятся на абстракции высокоуровневого языка. Именно устранение этой несогласованности является задачей смешанной семантической модели, представленной Шадриным в его диссертации [9].

Автор отмечает, что традиционные компиляторы трактуют машинный код как отдельный слой исполнения со своей моделью стека, регистрами и механизмами управления, что приводит к разрыву семантики при совмещении его с высокоуровневым представлением программы. Для устранения этого разрыва он вводит *единое промежуточное представление*, которое одновременно поддерживает исполнение как высокоуровневых программ, так и низкоуровневых инструкций.

В данном подходе переход между уровнями абстракций не требует смены модели исполнения и модификаций стека. Напротив, Шадрин определяет оба языка как две части единой абстрактной машины. Исполнение низкоуровневой инструкции интерпретируется как изменение того же состояния, которое используется и для высокоуровневых операторов. Схема взаимодействия двух уровней абстракций представлена на рисунке 5.

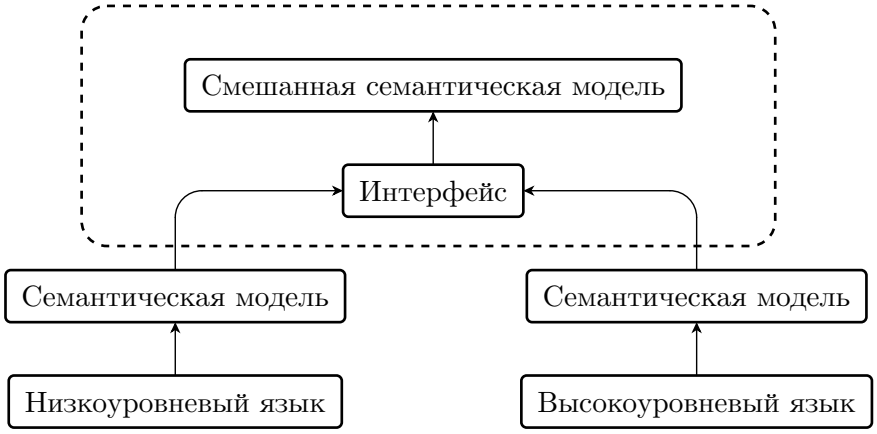


Рисунок 5 — схема взаимодействия семантических моделей разных уровней. Пунктирной областью выделен уровень, на котором происходит взаимодействие двух моделей

Именно за счёт такой интеграции становится возможным корректное взаимодействие разных моделей исполнения. Низкоуровневый код больше не нарушает абстракции высокоуровневого языка, высокоуровневые компоненты могут ссылаться на значения, используемые ассемблерной вставкой, без необходимости восстановления контекста или дополнительных соглашений. Семантика программы остается целостной, поскольку две взаимосогласованные модели используют общее промежуточное представление.

3 Постановка задачи

Целью данной работы является исследование существующих решений и экспериментальная реализация поддержки ассемблерных вставок в оптимизирующем статическом компиляторе многоязыковой виртуальной машины *Huawei*.

Для достижения поставленной цели в работе предполагается решить следующие задачи:

- изучить существующие подходы к реализации вставок в современных компиляторах и их ограничений;
- разработать экспериментальную реализацию поддержки вставок ассемблерного кода на базе статического компилятора многоязыковой виртуальной машины *Huawei*;
- применить реализованный механизм к уже существующему ассемблерному кодом в рамках данной виртуальной машины;
- интегрировать новый механизм ассемблерных вставок с существующими оптимизациями и анализами в компиляторе;
- исследовать подходы к анализу и оптимизации ассемблерных вставок и реализовать их;
- разработать набор тестов и бенчмарков для оценки эффективности решения;
- провести апробацию результатов исследования.

4 Дизайн и реализация

4.1 Описание окружения

Рассматриваемый статический компилятор является частью виртуальной машины *Huawei*, среда исполнения которой реализована на языке программирования *AJ* — специализированном языке системного программирования, предоставляющим как высокоуровневые, так и низкоуровневые средства, включая адресную арифметику.

Язык AJ является расширением языка Java при помощи аннотаций над классами и низкоуровневых концепций. Исходный код программы на языке AJ сначала транслируется в *Java байт-код*[7] с помощью специального транслятора *aj-javac*, который является модификацией *javac* – стандартного транслятора Java программ в Java байт-код.

Одна часть языковых конструкции языка AJ выражается через стандартные конструкции Java байт-кода, а другая часть остается представленной в виде внутренних методов (англ. *intrinsic*), реализация которых выполняется на последующих этапах компиляции программы.

В исходной реализации статического оптимизирующего компилятора виртуальной машины *Huawei* ассемблерные вставки представлялись в виде внешних методов. Реализовывались такие методы двумя способами:

[7]Байт-код Java — набор инструкций, исполняемых виртуальной машиной Java.

а) ассемблерный код написан с использованием синтаксиса определенных ассемблеров и размещался в отдельных файлах, передаваемых компоновщику[8] ;

б) требовалась отдельная поддержка вставок в инфраструктуре компилятора.

Данные подходы приводили к ряду ограничений, связанных с интеграцией ассемблерного кода в общий процесс компиляции:

– зависимость от внешних ассемблеров;

– ограниченность в применения оптимизаций и статического анализа компилятора;

– использование фиксированных регистров, усложняющее взаимодействие с системой распределения регистров;

– требование поддерживать каждую ассемблерную вставку в компиляторе по отдельности.

Поэтому возникла необходимость в механизме, позволяющем интегрировать ассемблерные вставки в основную инфраструктуру компиляции и преодолеть данные ограничения.

4.1.1 Внутренние методы

Внутренние методы (*англ. intrinsic*) — это методы, реализация которых задаётся не в исходном коде программы, а непосредственно в компиляторе. Схема обработки обычных методов и intrinsic-методов представлена на рисунках 6, 7 соответственно:

---

[8]Компоновщик (англ. linker) — программа, объединяющая объектные файлы и модули из библиотеки для создания исполняемого файла.

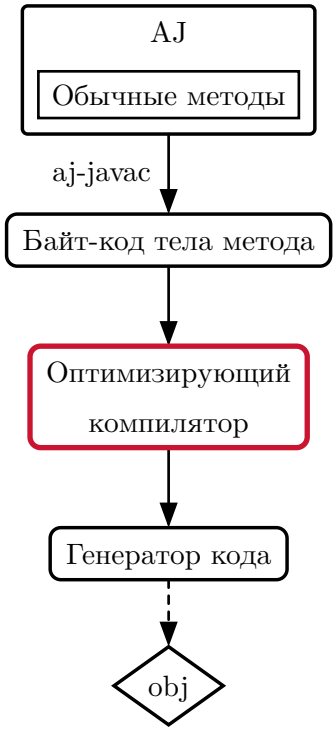


Рисунок 6 — обработка обычного метода

Рисунок 7 — обработка внутреннего метода

С точки зрения программы внутренние методы выглядят как обычные методы с заданной сигнатурой, однако их тело отсутствует в байт-коде и подставляется компилятором на этапе генерации промежуточного представления или машинного кода. Использование таких методов обусловлено следующими причинам:

– повышение производительности, например, генерация более оптимального кода для конкретной архитектуры. Зачастую это математические функции, работа с массивами;
– невозможность выразить семантику функции средствами языка высокого уровня, например, операции, связанные с аллокацией объектов;

– необходимость в более точном контроле над аппаратными ресурсами (например, барьеры памяти);

Для решения проблем, связанных с реализацией вставок в компиляторе *Huawei VM*, был использован подход, основанный на обзоре реализаций вставок в *HotSpot JVM* и *GraalVM*.

Основная идея заключается в представлении ассемблерных вставок как intrinsic-методов, тело которых сгенерируется специализированным генератором. Это позволяет достаточно просто интегрировать ассемблерные вставки в общий процесс компиляции.

4.1.2 Промежуточное представление программы

Промежуточное представление программы в рассматриваемом компиляторе имеет форму *Sea of Nodes* — графовой структуры, в которой операции и значения представлены вершинами, а зависимости между ними задаются рёбрами данных, управления, памяти.

Основным преимуществом данной формы представления является гибкость размещения операций в графе. В отличие от классических графовых представлений, например, графа потока управления (*CFG*), вершины в форме *Sea of Nodes* могут не иметь фиксированное положение. Вместо этого для каждой вершины определяются нижняя и верхняя границы, между которыми она должна быть расположена.

Выделяют два вида вершин: плавающие (*англ. floating*), которые не зависят от потока управления и данных, и фиксированные (*англ. spinal*) — вершины, привязанные к определённой позиции в графе.

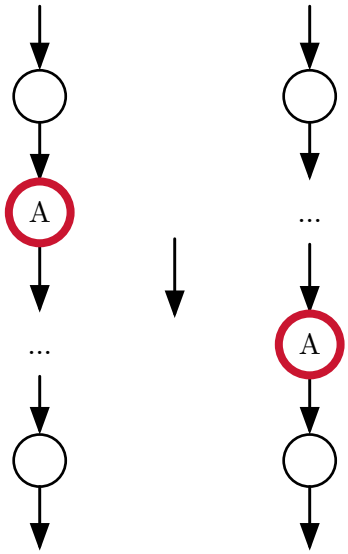


Рисунок 8 — перемещение плавающей вершины A вниз в графе потока управления

Плавающие вершины могут свободно перемещаться в графе, что упрощает проведение многих анализов и оптимизаций. Например, вынос инвариантных вычисление из циклов не требует какого-либо дополнительного анализа, так как выполняется неявно сразу при построении графа.

4.2 Классификация вставок в компиляторе

Интеграция ассемблерных вставок в *Sea of Nodes* позволяет рассматривать их как обычные узлы графа, на которые распространяются стандартные механизмы анализа и оптимизации.

Вставки, представленные в виде плавающих вершин, не имеют зависимостей по управлению и оперируют только потоками данных. В отличие от них, вставки с фиксированным положением в графе обладают побочными эффектами (например, записью в память).

Рисунок 9 — вставка в виде плавающей вершины, красные дуги обозначают поток данных

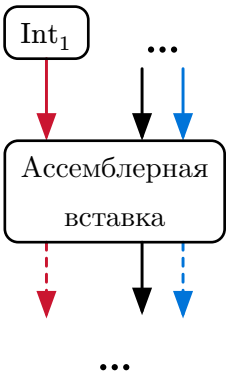


Рисунок 10 — вставка в виде фиксированной вершины, красные дуги обозначают поток данных, чёрные – поток управления, синие – зависимость по памяти

В действительности разделение вставок является более глубоким и не ограничивается определением зависимостей вершины, поскольку непосредственно связано с генерацией кода.

Так, ассемблерные вставки в компиляторе можно классифицировать на три категории по ключевым свойствами, представленными в таблице 1.

Таблица 1 — категоризация ассемблерных вставок на основании их свойств

Категория / Свойства	Внешние вставки	Фиксированные вставки	Плавающие вставки
Имеют побочные эффекты	Да	Да	Нет
Используют только фиксированные регистры	Да	Могут быть как фиксированные, так и произвольные	Нет
Самостоятельно формируют стековый кадр	Да	Нет	Нет

В большинстве случаев, и фиксированные, и плавающие ассемблерные вставки используют переданные им аргументы, соблюдают соглашение о вызовах и не выполняют явного управления стеком. Для них достаточно декларативно указать ограничения на регистры, после чего компилятор корректно учтет их при распределении ресурсов.

Однако существуют фиксированные вставки, которые обладают свойствами, отличными от упомянутых выше. Будем называть такие вставки *внешними*.

Внешние вставки могут изменять указатель стека, изменять состояние памяти, сохранять и восстанавливать конкретные регистры «вручную» и формировать собственный стековый кадр.

Такие вставки не могут быть адаптированы механизмами оптимизирующего компилятора и должны генерироваться в соответствие с заданной семантикой. По этой причине подобные вставки выделяются в отдельную категорию и обрабатываются как внешние вызовы (рисунок 11).

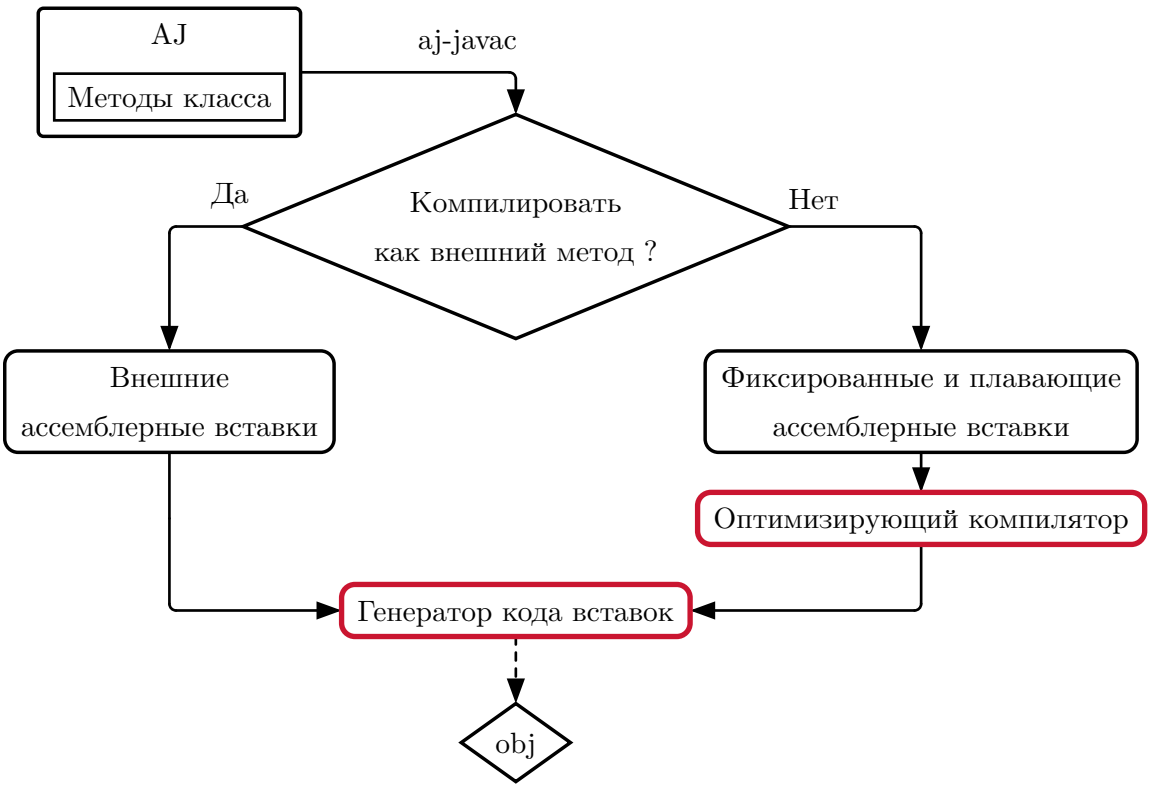


Рисунок 11 — инфраструктура компилятора с точки зрения вставок

Примером таких вставок является *TrapHandler* — внутренний метод, который перенаправляет вызов в обработчик исключения. Для него важен доступ к определенным регистрам и работа с указателем на стек: он вызывает конкретный обработчик, передавая ему аргументы на фиксированных регистрах, поэтому его тело должно быть сгенерировано именно так, как оно определено.

Похожей задачей занимается *ContextUtils*, который сохраняет контекст текущего потока на стек, смещает указатель на стек и восстанавливает контекст.

Фиксированные вставки могут быть адаптированы компилятором, поскольку могут иметь произвольные регистры для входных и возвращаемого значения, однако они имеют побочные эффекты, например, запись в память.

Примером фиксированной вставки является реализация атомарной операции *CompareAndSwap*, принимающая три аргумента, второй из которых должен быть расположен на регистре `rax`. Эта вставка неявно меняет значение в памяти, а потому имеет строго определенное положение в графе потока управления.

Плавающие вставки — самые подвижные в графе. Поскольку у них нет зависимости от памяти и потока управления, компилятор может свободно применять к ним оптимизации по перемещению и трансформации в графе.

Примером такой вставки является *BitSwap* — внутренний метод, который принимает аргумент на произвольном регистре общего назначения или вещественном регистре (в зависимости от типа данных) и инвертирует порядок бит в его представлении.

4.2.1 Выражение классификации в компиляторе

В целом, при трансляции байт-кода программы в промежуточное представление необходимо задать отображение. Это отображение реализуется на основании имени класса, в котором определен данный метод, имени метода и сигнатуры в виде байт-кода и некоторой опциональной информации.

Для генерации вершины ассемблерной вставки необходима информация о ее категории. Поэтому дополнительной метаинформацией при отображении метода были введены булевые характеристики *hasNativeWrapper* для обозначения внешних вставок и *isFloatingNode* для

плавающих (фиксированные вставки имеют значение данного параметра *false*). Значения характеристик для вставок приведены в таблице 2.

Таблица 2 — значения характеристик для выделенных категорий вставок

Характеристика / Категория вставок	*hasNativeWrapper*	*isFloatingNode*
Внешние	true	false
Фиксированные	false	false
Плавающие	false	true

Для представления вставки в промежуточном представлении были реализованы следующие классы вершин: абстрактный класс *AsmIntrinsic* и его наследники *AsmIntrinsicFloating* и *AsmIntrinsicSpinal* для плавающих и фиксированных вставок соответственно. В случае внешних вставок так же генерируется *AsmIntrinsicSpinal*, но трансформации к этому узлу не применяются.

4.3 Интерфейс ассемблерных вставок

Основываясь на свойствах выделенных вставок, можно формально определить то, какие аспекты необходимо учесть при разработке интерфейса.

4.3.1 Соглашение о вызовах

Выполнение любой функции происходит в рамках соглашения о вызовах, которое декларирует способ передачи аргументов, возврата результата, и правила использования регистров и организацию стекового кадра.

Стековый кадр представляет собой область памяти, выделяемую на стеке для конкретного вызова функции. Он содержит аргументы, локальные переменные, сохранённые значения регистров и служебную информацию, необходимую для возврата управления (рисунок 12).

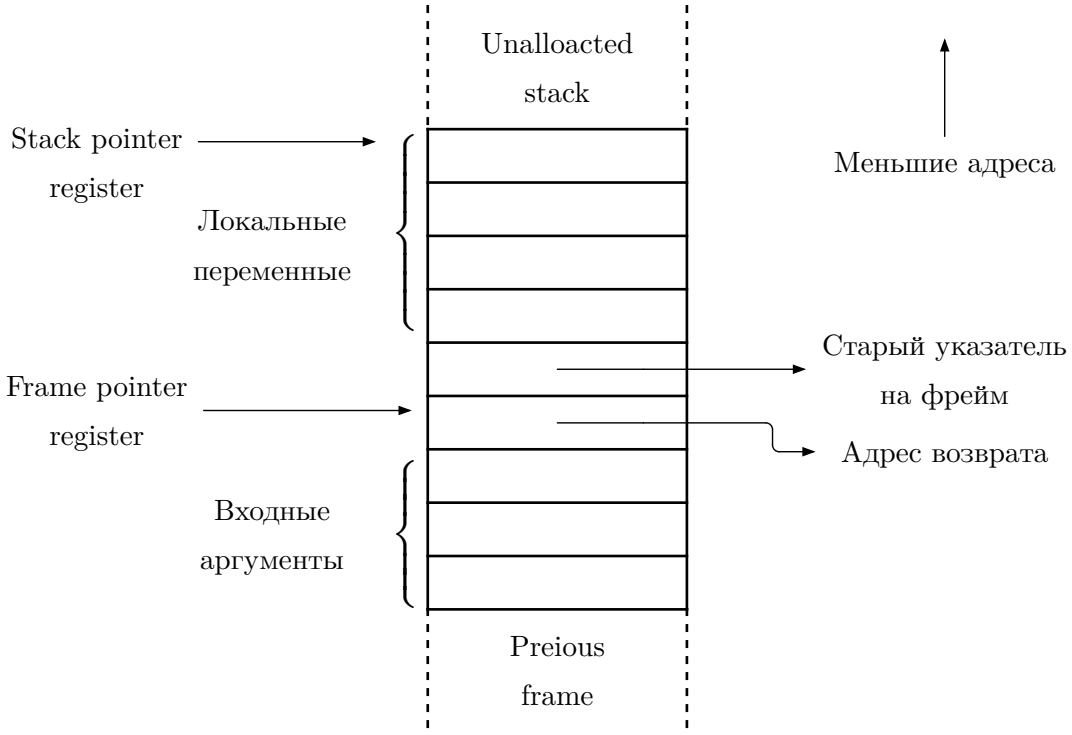


Рисунок 12 — стандартная раскладка стекового кадра

Поскольку ассемблерная вставка реализована в компиляторе в виде внутреннего метода, она так же должна соблюдать контракт, описанный соглашением о вызовах. Поэтому возникает необходимость в описании того, как она будет взаимодействовать с окружением компилятора.

4.3.2 Распределение регистров

В частности, соглашение о вызовах определяет то, какие регистры могут быть использованы. Поскольку число регистров ограничено, возникает задача их распределения. Эта задача решается компилятором с помощью механизма распределения регистров.

При этом ограничения, задаваемые соглашением о вызовах, напрямую учитываются системой распределения регистров при построении отображения значений на регистры и память. Следовательно, при интеграции ассемблерных вставок компилятору необходимо предоставить дополнительную информацию о том, какие регистры используются вставкой, чтобы не нарушить корректность распределения.

В рассматриваемом оптимизирующем компиляторе уже реализован механизм распределения регистров, отвечающий за построение искомого отображения, а необходимые ограничения на регистры представляются в виде множеств.

Множества регистров формируются на основании ABI, задающего набор регистров, доступных на целевой архитектуре. Регистры при этом разделяются по типу хранимых значений: основными являются целочисленные и вещественные регистры. Будем обозначать соответствующие множества как IRegs и FRegs.

Для простоты передачи информации об ограничения, было предложено абстрактное представление *LocationDesc* (листинг 13).

```
// Описывает множества регистров для аргументов
sealed abstract class LocationDesc
case class Not(loc: Location) extends LocationDesc
case class Only(loc: Location) extends LocationDesc
case class All extends LocationDesc
```

Листинг 13 — базовый класс LocationDesc и его реализации

Это представление задает отображение для целочисленных (для вещественных аналогично) регистров по следующим правилам:

$$\text{Not(R)} \to \text{IRegs}\,/\{\text{R}\} \qquad \text{Not(R)} \to \text{FRegs}\,/\{\text{R}\}$$

$$\text{Only(R)} \to \{\text{R}\} \qquad \text{Only(R)} \to \{\text{R}\}$$

$$\text{All} \to \text{IRegs} \qquad \text{All} \to \text{FRegs}$$

Важной особенностью является возможность точно не указывать, какие именно регистры используются в ассемблерной вставке. Для этого существует правило *All*, для которого соответствует множество всех доступных регистров.

4.3.3 Передача аргументов

Для корректного вызова ассемблерной вставки необходимо явно указать, где располагаются ожидаемые значения. В общем случае соглашение о вызовах допускает передачу аргументов как через регистры, так и через стек. Однако в рассматриваемой реализации основное внимание уделяется аргументам, передаваемым через регистры.

С точки зрения интерфейса ассемблерной вставки входные аргументы представляются как упорядоченный набор значений. Для каждого аргумента задаётся множество допустимых регистров, в которых он может быть размещён.

Задаваемое отображение схематично изображено на рисунке 13.

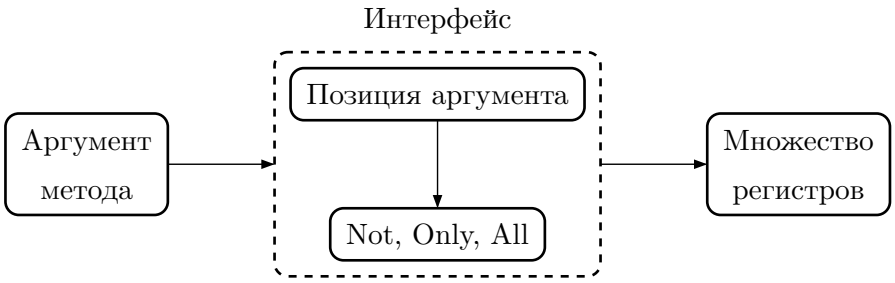


Рисунок 13 — схема отображения аргументов вставки во множество регистров

Это отображение строится с помощью абстрактного представления *LocationDesc*, описывающего ограничения на регистры, и далее используется механизмом распределения регистров. Компилятор выбирает конкретные регистры, удовлетворяющие всем ограничениям, и размещает в них значения перед выполнением вставки.

Например, если операция требует, чтобы первый аргумент находился в конкретном регистре (например, *rax*), это может быть выражено как $arg_0 \rightarrow Only(RAX)$. Если же допустим любой регистр, используется $arg_0 \rightarrow All$.

4.3.4 Возвращаемое значение

Вызовы методов могут иметь возвращаемое значение либо не иметь его (*void*). Описание возвращаемого значения напрямую влияет на процесс распределения регистров и должно быть согласовано с соглашением о вызовах.

Так, возвращаемое значение должно быть размещено в строго определенных регистрах. Например, для архитектуры *x86-64* результат целочисленной операции обычно возвращается в регистре *rax*.

Нередко результат ассемблерной вставки располагается на регистре, совпадающем с одним из регистров аргументов. Данную информацию можно заранее сообщить компилятору для того, чтобы распределение регистров было проведено более эффективно.

Связывание регистра с результатом задаётся декларативно с использованием следующей абстракции *ResultLocationDesc* (листинг 14).

```
// Описывает множества регистров для возвращаемого значения
sealed abstract class ResultLocationDesc
case class Bound(idx:Int) extends ResultLocationDesc
case class Unbound(loc: LocationDesc) extends ResultLocationDesc
case class NoLocation extends ResultLocationDesc
```

Листинг 14 — базовый класс ResultLocationDesc и его реализации

Здесь *Bound(idx)* используется для связывания регистра аргумента на позиции *idx* с результатом. Если же возвращаемое значение не связано с аргументами, то используется *Unbound(LocationDesc)*, где множество допустимых для размещения результатов регистров описывается при помощи предметно-ориентированного языка *LocationDesc*.

Примером вставки, результат которой связан с определенным аргументом, является атомарная операция *CompareAndSwap*. В силу того, что инструкция *cmpxchg* ожидает второй аргумент на регистре *rax* и размещает результат на нем же, можно явно указать *Bound(1)*.

4.3.5 Волатильные ресурсы

Определение 7. *Волатильные ресурсы (англ. volatiles) — ресурсы, значения в которых могут быть изменены неявно как побочный эффект*

работы инструкции или в соответствии с соглашением о вызовах (ABI).

К волатильным относятся регистры, значения в которых перезаписываются инструкцией неявно или используются для передачи аргументов и возврата результата. Также к этой категории относятся регистры, которые согласно соглашению о вызовах классифицируются как не сохраняемые вызываемой стороной (англ. caller-saved).

Указание волатильных регистров необходимо для корректной работы механизма распределения регистров: компилятор должен либо не размещать в них важные значения, либо сохранить их перед выполнением вставки.

Примером волатильных ресурсов могут являться регистры, используемые математическими операциями (*IDiv, UDiv, UMulH, IRem, URem*) — *rax, rdx* используются для размещения результата и промежуточных данных. Для атомарной операции *CompareAndSwap* — волатильным является регистр *rax*, используемый инструкцией *cmpxchg* для сравнения и сохранения результата.

4.3.6 Временные ресурсы

Определение 8. *Временные ресурсы (англ. temporals) — ресурсы, используемые внутри ассемблерного блока для хранения промежуточных результатов вычислений.*

В отличие от волатильных, временные ресурсы не имеют фиксированной семантики, поскольку они используются исключительно для хранения временных значений.

С точки зрения распределения регистров это означает, что компилятор не будет использовать его для размещения входных и выходных операндов данной вставки. Это делает ресурс доступным для свободного использования внутри ассемблерного кода без риска повредить данные программы.

Примером использования временных регистров является реализация барьеров чтения в алгоритме сборки мусора. Такие барьеры перехватывают обращения к полям объектов, которые могут быть перемещены во время конкурентной эвакуации. Для реализации данного механизма используется один или два временных регистра в зависимости от свойств аргумента.

4.3.7 Тело ассемблерной вставки

Одним из существенных ограничений при интеграции вставок является использование внешних ассемблеров (например, *GNU Assembler*). Тело вставки становится зависимым от конкретного синтаксиса ассемблера.

Для решения этой проблемы предлагается использование предметно-ориентированного языка, предоставляемого компилятором. Например, для статического компилятора *Huawei VM* используется *AsmEmitter*. Для *HotSpot JVM* таким языком может выступать *MacroAssembler*.

Так, на примере ниже используется *AsmEmitter* для написания цикла.

```
val loop = asm.newLabel ; Создаем объект метки
asm.bind(loop)          ; Объявляем (связываем) метку в коде
; Логика цикла
asm.sub(counter, 1)
asm.jcc(NZ, loop)       ; Проверка на неотрицательное значение (jnz)
```

Листинг 15 — пример цикла с использованием AsmEmitter

Предметно-ориентированный язык позволяет выражать низкоуровневые конструкции с использованием высокоуровневого языка. Зачастую это упрощает процесс разработки и сокращает объем кода за счет использования стандартных управляющих конструкций, таких как циклы или вызовы функций (рисунок 16).

```
; Описание логики на DSL            Результат генерации (байты инструкций)
val regs    = {R1, ..., Rn}         ; mov [0xbebebe + 0], R1
val base    = 0xbebebe              ; mov [0xbebebe + 0], R1
for i in 0..n {                     ; ...
  DSL.mov(base + i * 8, regs(i))    ; mov [0xbebebe + (n-1)*8], Rn
}
```

Листинг 16 — псевдокод генерации последовательности инструкций через цикл с использованием абстрактного предметно-ориентированного языка DSL

Такой подход так же позволяет обобщить некоторые реализации вставок между разными архитектурами, например вставки, сохраняющие и восстанавливающие контекст текущего потока (приложение А).

4.3.8 Генерация машинного кода

Подход с использованием внутреннего предметно-ориентированного языка в компиляторе позволяет генерировать машинный код напрямую, минуя зависимости от внешних утилит вроде ассемблера.

На этом этапе абстрактные команды используемого языка преобразуются в конкретные последовательности байт, формируя структуру объектного файла, который в процессе генерации, компоновки и загрузки машинного кода выступает в роли промежуточного представления программы.

Опишем в общем виде процесс генерации машинного кода. Так, объектный файл можно рассматривать как набор сегментов, представимых в виде тройки

$$S = \langle B, L, F \rangle,$$

где B – последовательность байт, L – множество меток, задающих отображение в позицию внутри сегмента, а F – множество ссылок для соответствующего сегмента.

Ссылка (англ. fixup) представляет собой описание места в коде, требующего уточнения значения, и может быть формально определена как четверка

$$f = \langle p, t, o, k \rangle,$$

$p \in \mathbb{N}$ – позиция в сегменте, t – целевой объект (метка или внешний символ), $o \in \mathbb{Z}$ – дополнительное смещение, а k — тип ссылки, определяющий способ кодирования и размер значения.

Тип ссылки определяет способ кодирования и размер значения. Существуют, например, абсолютная 32-битная ссылка, относительная 32-битная ссылка, используемая в инструкциях перехода, ссылка, кодирующая младшие 12 бит адреса в инструкциях загрузки констант.

Сегменты содержат ссылки на *внутренние* и *внешние* символы относительно данной бинарной компоненты. Значение таких ссылок не всегда известно на определенном этапе генерации, поэтому возникает задача их разрешения.

Ссылки, которые не могут быть разрешены на данный момент, преобразуются в *записи о релокации* и сохраняются в объектном файле и используются на этапе компоновки.

Способ представления записей о релокации, а также кода, данных и символов определяется форматом объектного файла. Форматы объектных файлов являются платформенно-зависимыми. Так, к распространенным форматам относятся, например, COFF и PE[9] для платформы Windows. В Unix-подобных системах стандартным форматом объектных файлов является формат ELF[10].

Следует отметить, что разрешения ссылок возможно проводить на более ранних этапах компиляции программы. Так, в компиляторе виртуальной машины *Huawei* похожий механизм используется непосредственно в процессе генерации кода. Представление бинарной

---

[9]Microsoft Build 2026. PE file format // Microsoft Learn. 2026. URL: https://learn.microsoft.com/ru-ru/windows/win32/debug/pe-format (дата обращения: 12.03.2026).

[10]Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification. Version 1.2 / TIS Committee. 1995. URL: linuxfoundation.org (дата обращения: 14.05.2026).

компоненты в виде набора сегмента позволяет разделить разрешение ссылок на несколько уровней (рисунок 14).

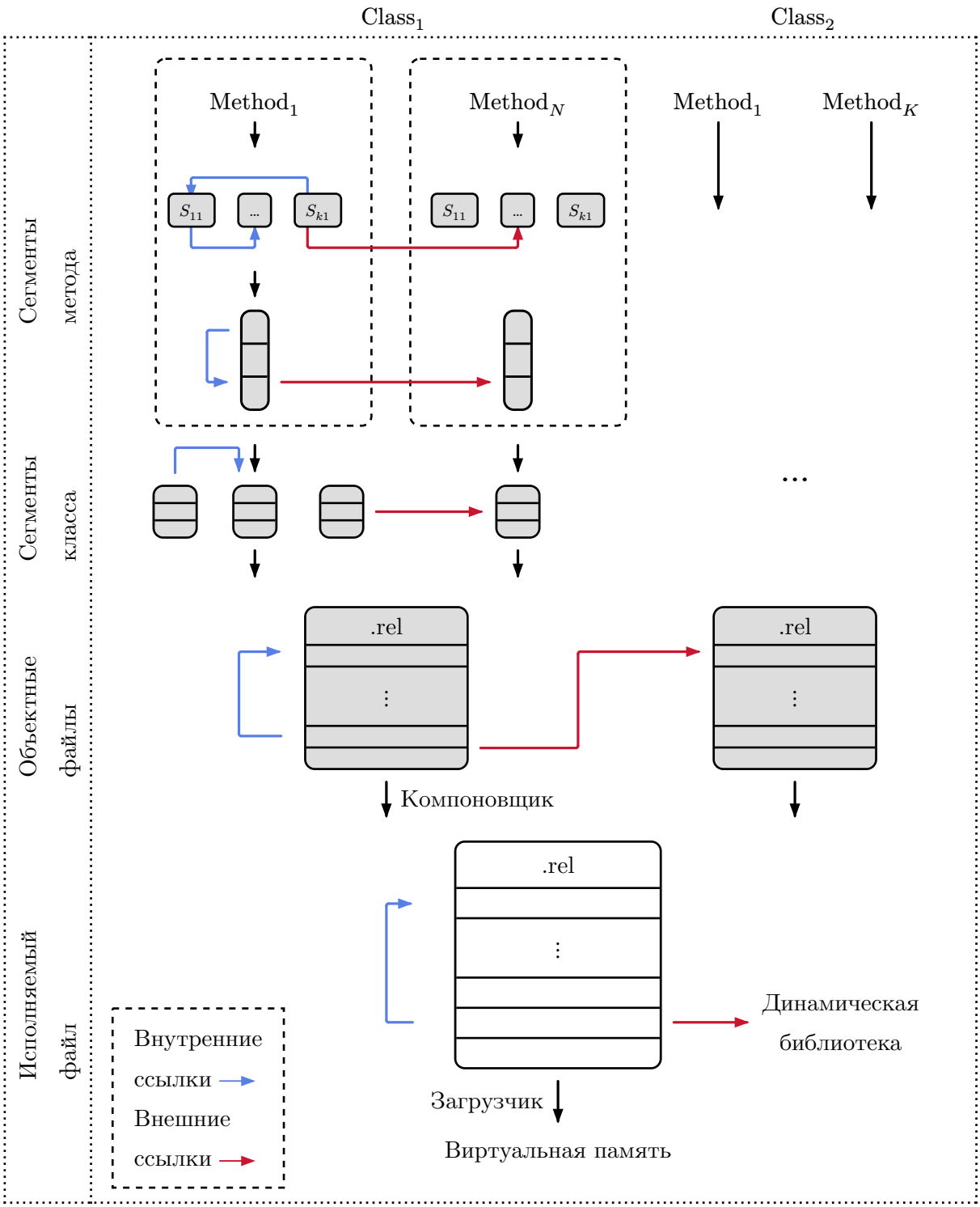


Рисунок 14 — схема разрешения ссылок по уровням: метод, класс, объектный и исполняемый файлы

Процесс начинается на уровне отдельных методов, где формируются локальные сегменты кода S_{ik}. Данные сегменты S_{ik} объединяются для конкретных методов. Затем сегменты отдельных методов формируют сегмент кода класса, которому они принадлежат.

По мере продвижения по уровням от методов к классам и объектным файлам – компилятор и компоновщик разрешают доступные ссылки. Если часть ссылок остается неразрешенной (например, при использовании динамических библиотек), соответствующие релокации сохраняются для этапа загрузки. Загрузчик[11] отображает сегменты исполняемого файла в виртуальную память процесса, выполняет окончательное размещение сегментов в памяти и применяет оставшиеся релокации, используя фактические адреса загруженных модулей.

4.3.9 Код интерфейса ассемблерных вставок

Для интеграции ассемблерной вставки в инфраструктуру компилятора был представлен интерфейс, который обеспечивает корректное взаимодействие вставки с механизмами компилятора.

Добавление конкретной вставки требует реализации интерфейса, который был реализован на языке *Scala*. Код представлен на листинге 17.

[11]Загрузчик (англ. loader) — часть операционной системы, которая отвечает за загрузку программы (и библиотек) в память и подготавливает их к исполнению.

```
trait IntrinsicGenerator {
  def argsLocation:   Seq[LocationDesc] = Seq()
  def resultLocation: ResultLocation = NoLocation
  def volatileIRegs:  Seq[Location] = Seq()
  def volatileFRegs:  Seq[Location] = Seq()
  def temporalsAmountIRegs: Int = 0
  def temporalsAmountFRegs: Int = 0

  // генерация внешней вставки (не принимает аргументы)
  def genBody(): Unit = ()

  def genBody(paramRegs: Seq[Location],
              resReg: Option[Location],
              tempIRegs: Seq[Location],
              tempFRegs: Seq[Location]): Unit

  // перезагрузка genBody(...) для различных реализаций,
  // например, для возвращаемого значения void или отсутствующих tempIRegs
  ...
}
```

Листинг 17 — интерфейс генератора ассемблерных вставок IntrinsicGenerator

Опишем данный интерфейс:

а) Входные аргументы (4.3.3) задаются в виде последовательности, состоящей из элементов типа *LocationDesc*. Для этого необходимо переопределить метод *argsLocation*.

б) Информация о результирующем значении (4.3.4) указывается через метод *resultLocation*, который возвращает значение *ResultLocation*.

в) Для указания сведений об волатильных регистрах (4.3.5) используется метод *volatuleIRegs* (*volatileFRegs*

для веществennозначных регистров), возвращающий последовательность элементов типа *Location*.

г) В случае временных регистров необходимо указать их количество (*temporalsAmountIRegs, temporalsAmountFRegs*).

д) Логика вставки реализуется через переопределение *genBody(…)*, в случае внешних вставок должен быть переопределен метод *genBody()*, поскольку внешние вставки не адаптируются оптимизирующим компилятором. Иначе переопределяется один из возможных методов *genBody(…)* в зависимости от требований ассемблерной вставки. Например, некоторые вставки имеют возвращаемое значение *void*, поэтому для них должен быть переопределен метод, который не принимает *resReg*.

4.3.10 Пример использования

Рассмотрим пример того, как был применен интерфейс к ассемблерной вставке атомарной операции *CompareAndSwapLong* в оптимизирующем компиляторе, используемой для неблокирующей синхронизации.

Данная вставка представлена в виде внешнего метода *CompareAndSwapLong* на языке *AJ* (листинг 18). Этот метод имеет три аргумента — *addr, expectedValue, newValue* и возвращает значение типа *long* (листинг 19).

```
@External
private native long compareAndSwapLong(Address addr,
                                       long expectedValue,
                                       long newValue);
```

Листинг 18 — внутренний метод compareAndSwapLong в среде поддержки исполнения на языке AJ

Семантика метода следующая: атомарно сравнивается содержимое ячейки памяти (*addr*) с заданным (*expectedValue*) значением и, только если они совпадают, изменяет содержимое этой ячейки памяти на новое заданное значение (*newValue*).

Тело вставки (листинг 19) реализовано в отдельном ассемблерном файле для платформы *x86-64*.

```
CompareAndSwapLong:
  ; rcx = addr
  ; rsi = expectedValue
  ; rdx = newValue
  mov rax, rsi
  lock cmpxchg qword [rcx], rdx
  mov rcx, rax
  ret
```

Листинг 19 — ассемблерная вставка CompareAndSwap

Используемое соглашение о вызовах обязует располагать первые три аргумента на регистрах *rcx, rsi, rdx*, а возвращаемое значение – на регистре *rcx*. Поэтому вставка содержит две пересылки данных: размещение *expectedValue* на регистре *rax*, так как инструкция *cmpxchg* неявно использует этот регистр, и размещение результата с *rax* на *rcx*.

Рассмотрим то, как стала выглядеть реализация вставки.

Для внутреннего метода в AJ была добавлена аннотация *@Intrinsic*. В промежуточном представлении данная вставка является фиксированной, поэтому для генерируется вершина *AsmIntrinsicSpinal*. Реализация вставки через интерфейс представлена на листинге 20.

```
class CASLongGenerator extends IntrinsicGenerator {

  override def argsLocation   = Seq(Not(RAX), Only(RAX), Not(RAX))
  override def resultLocation = Bound(1)
  override def volatileIRegs  = Seq(RAX)

  // Процесс сопоставления выделенных ресурсов с аргументами
  override genBody(paramRegs: Seq[Location]): Unit = paramRegs match {
    ...  => gen(addr, newVal)
  }

  private def gen(addr: AddrMode, newVal: Register): Unit = {
    asm.lock();
    asm.cmpxchg(addr, newVal.as(tpe.width))
  }
}
```

Листинг 20 — интеграция вставки CompareAndSwapLong через предложенный интерфейс

а) Поскольку логика вставки требует размещение второго аргумента на регистре *rax*, то можно сразу сообщить компилятору об этом требовании. При этом необходимо исключить *rax* из доступных регистров для оставшихся аргументов.

б) Результирующее значение располагается на регистре *rax*, поэтому его можно связать с регистром второго аргумента

с использованием абстрации *ResultLocationDesc*. В данном примере для этого используется *Bound(1)*.

в) Для обозначения *rax* волатильным регистром используется переопредленный метод *volatileIRegs*.

г) Переданная информация об аргументах, возвращаемом значении и волатильных регистрах позволяет делегировать компилятору вставку пересылок данных и терминирующей инструкции *ret*, оставив только основную логику вставки.

4.3.11 Применение оптимизаций

Представленный подход к реализации вставок ассемблерного кода позволяет применять множество оптимизаций благодаря интеграции вставок в промежуточное представление *Sea of Nodes*.

4.3.11.1 Перемещение плавающих вставок

Для низкоуровневых этапов компиляции, таких как кодогенерация и распределение регистров, необходимо закрепить вершины промежуточного представления *Sea of Nodes* в графе потока управления на определенной позиции.

Линеризация графа реализуется при помощи алгоритма *Global Code Motion* [21] (GCM), реализованного в рассматриваемом компиляторе. Данный алгоритм перемещает вершины на позиции наиболее выгодные для генерации кода.

Например, этот алгоритм позволяет уменьшить давление на регистры и уменьшить количество вычислений засчет удаления неиспользуемого кода и свертки значений.

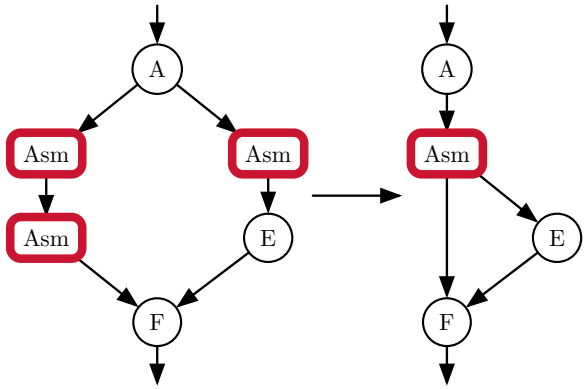


Рис. 15 — удаление дублирующегося кода и вынесение общего подвыражения (Asm)

Поэтому, если вставка не обладает побочными эффектами, компилятор может исключить избыточные вызовы или заменить их константными значениями, вынеся общее подвыражение (рисунок 15).

4.3.11.2 Распределение регистров

В контексте распределения регистров алгоритм GCM минимизирует количество одновременно активных переменных.

В совокупности с тем, что в предложенной реализации вставок для описания регистров аргументов используются допустимые множества, компилятор может размещать вставки более эффективно. Это позволяет снизить давление на регистры, избегая принудительного вытеснения данных в память, сократить число пересылок для сохранения и восстановления регистров.

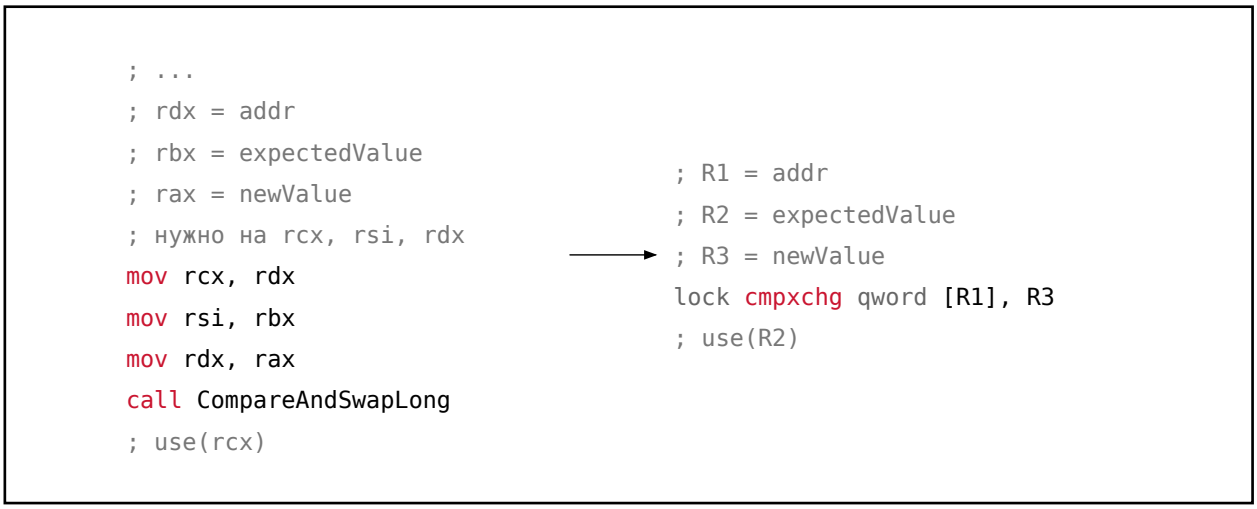
Примером может служить рассмотренная ранее реализация *CompareAndSwapLong* (листинг 20). Здесь использование конкретных регистров *rcx, rsi, rdx* было абстрагировано до множеств *Not(RAX), Only(RAX), Not(RAX).*

4.3.11.3 Открытая подстановка вызовов

Другим важным механизмом оптимизации является открытая подстановка (англ. inlining), при котором тело вызываемой функции подставляется непосредственно в точку вызова.

Это не только устраняет накладные расходы на вызов (сохранение контекста, передача аргументов), но и расширяет контекст для работы оптимизатора, позволяя GCM более эффективно перемещать узлы вставки внутри вызывающего метода.

Рассмотрим на примере фиксированной вставки *CompareAndSwapLong* (листинг 19), тело которой содержит пересылки данных на нужные регистры.



Листинг 21 — сгенерированный код до и после применения интерфейса

На листинге 21 показано, что использование интерфейса позволило избавиться от вызова внешнего метода. Компилятор смог оптимально разместить код вокруг вставки, что снизило давление на регистры и позволило избавить от пересылок данных.

4.4 Сравнение реализованного механизма с другими подходами

Предложенный механизм отличается от рассмотренных реализаций способом описания вставок и уровнем их интеграции в инфраструктуру компилятора.

В компиляторах *gcc* и *clang* ассемблерные вставки описываются в виде текста с использованием строковых ограничений. Такой подход позволяет задавать требования к размещению операндов, однако не интегрирует вставку в промежуточное представление программы и ограничивает возможности анализа и оптимизации.

В инфраструктуре компилятора *C2* виртуальной машины *OpenJDK HotSpot* вставки описываются через внешний архитектурно-специфичный формат AD-файлов. Этот механизм позволяет задавать ограничения на регистры и побочные эффекты, однако требует использования отдельного языка описания.

В *GraalVM* вставки интегрированы в низкоуровневое представление через специализированные классы LIR. Ограничения на размещение операндов задаются в коде построения низкоуровневого представления и требуют явного назначения регистров на этапе понижения промежуточного представления.

В предложенной реализации описание вставки задаётся через единый интерфейс, представленного в виде абстрактного класса, методы которого должны быть переопределены. Ограничения на размещение аргументов, связывание результата с входными операндами, количество временных ресурсов, набор волатильных ресурсов и тело вставки определяются на уровне интерфейса.

Дополнительно предложенный механизм вводит классификацию вставок на внешние, фиксированные и плавающие, что позволяет явно задавать степень их интеграции в инфраструктуру компилятора. Также, ассемблерные вставки представляются отдельными узлами промежуточного представления и участвуют в анализе зависимостей, планировании исполнения и оптимизациях.

Для вставок компилятор самостоятельно может вставлять пересылки данных, сохранение и восстановление регистров, формировать стековый кадр, как в случае других индустриальных компиляторов.

Ограничениями предложенного механизма являются отсутствие выражения побочных эффектов на флаги процессора, отсутствие автоматической верификации корректности заданных ограничений, а также непрозрачность семантики ассемблерных инструкций в теле вставки для оптимизатора. Последнее ограничение является общим для большинства существующих подходов.

5 Результаты

Корректность работы всех реализованных вставок была протестирована на представительном наборе тестов и приложений, включающий **SPECjbb2008, SPECjbb2015, DaCapo**.

С точки зрения скорости исполнения программ вклад вставок оказался незначительным, поэтому для оценки эффективности механизма были проведены замеры производительности на наборе микробенчмарков. Сравнивались два способа реализации вставок: через вызов внешнего метода и через специализированные вершины промежуточного представления. Рассматривались следующие ассемблерные вставки:

а) математические функции *exp*, *rint*, *floor*, *acos*, *asin*, *pow*;

б) побитовое преобразование *bitswap*;

в) атомарная операция *CompareAndSwap* (*CAS*).

Математические вставки и *bitswap* представляют особый интерес, поскольку относятся к плавающим вставкам. *CompareAndSwap* является фиксированной.

Для апробации результатов был представлен набор следующих бенчмарков:

а) *Loop-invariant elimination* — цикл из 1 миллиона итераций с инвариантными вычислениями. Тест оценивает возможность выноса вычислений из цикла, удаления мёртвого кода, удаления дублирующего кода и свертки констант.

б) *Register pressure* — цикл из 100 тысяч итераций с обработкой четырёх массивов. Тест оценивает влияние на количество межрегистровых пересылок и сохранений и восстановлений регистров.

Измерения проводились на *13th Gen Intel Core i7-13700, 24 CPUs, 32 GB RAM*. В таблице 3 приведены средние значения времени выполнения и стандартное отклонение в миллисекундах.

Таблица 3 — средние значения времени выполнения и стандартное отклонение в миллисекундах

Название бенчмарка / Ассемблерная вставка	Loop-invariant elemination	Register pressure
math-intrinsics-native	3502ms ± 30ms	2818ms ± 2ms
math-intrinsics-asm	4.6ms ± 0.2ms	1954ms ± 4ms
bitswap-native	120ms ± 2ms	2297ms ± 2ms
bitswap-asm	4.7ms ± 2ms	2175ms ± 4ms
cas-native	2459ms ± 18ms	7357ms ± 36ms
cas-asm	2442ms ± 20ms	7353ms ± 24ms

Для удобства сравнения результатов в таблице 4 приведено относительное ускорение по времени выполнения:

Таблица 4 — ускорение времени выполнения тестов в X-раз

Название бенчмарка / Ассемблерная вставка	Loop-invariant elemination	Register pressure
math-intrinsics	761.3 раз	1.44 раза
bitswap	25.53 раза	1.06 раза
cas	1.01 раза	1.001 раз

Наибольший прирост производительности наблюдается для математических вставок в тесте *Loop-invariant elimination*, где ускорение составило **761 раза**, и *bitswap* — ускорение составило **25.5 раза**.

Такой результат объясняется тем, что после представления вставки в виде плавающей вершины компилятор получил возможность выполнить вынос инвариантных вычислений из тела цикла и произвести последующее удаление мёртвого кода.

Для операции *CAS* в тесте *Loop-invariant elimination* существенного прироста производительности не наблюдается, поскольку вставка является фиксированной. Помимо этого, время исполнения так же объясняется стоимостью выполнения атомарной инструкции *lock cmpxchg*, которая генерируется после открытой подстановки.

В тесте *Register pressure* для *math-intrinsic* ускорение составило **1.44 раза**. Для *bitswap* и *CAS*, несмотря на незначительное влияние на время исполнения, наблюдается сокращение размера сгенерированного кода за счёт устранения инструкций сохранения и восстановления регистров, а также уменьшения количества межрегистровых пересылок (листинги Б.27, Б.28, Б.25, Б.26).

ЗАКЛЮЧЕНИЕ

В результате данной работы были выполнены следующие задачи:

а) изучены существующие подходы к реализации вставок в современных компиляторах и их ограничения;

б) реализован и интегрирован механизм генерации ассемблерных вставок в оптимизирующий компилятор виртуальной машины *Huawei*;

в) проанализированы и переписаны **32** вставки, удалось обобщить 3 реализации вставки, переписано 3152 строки ассемблерных файлов, уменьшение числа строк на **17%**;

г) предложенный механизм интегрирован с оптимизациями и анализами;

д) корректность работы протестирована на представительном наборе тестов и приложений;

е) предложенный механизм демонстрирует улучшение производительности программ благодаря интеграции с существующими оптимизациями в компиляторе.

Дальнейшие исследования

а) интеграция в язык программирования AJ в виде аннотаций;

б) верификация ассемблерной вставки на основе обзора предшествующих работ;

в) расширение возможностей интерфейса.

ПУБЛИКАЦИИ

1. Васько М.Б. Эффективная реализация ассемблерных вставок в компиляторе языка высокого уровня. // Материалы 64-й Международной научной студенческой конференции МНСК-2026: Математика. Новосиб. гос. ун-т. Новосибирск, 2026.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Using the GNU Compiler Collection (GCC) - 6 Extensions to the C Language Family - 6.45 How to Use Inline Assembly Language in C Code [Электронный ресурс]. URL: https://gcc.gnu.org/onlinedocs/gcc/Using-Assembly-Language-with-C.html# (дата обращения: 03.10.2025).
2. LLVM Language Reference: Inline assembly expressions [Электронный ресурс]. URL: http://llvm.org/docs/LangRef.html#inline-assembler-expressions (дата обращения: 03.10.2025).
3. Microsoft C++ Inline Assembler Documentation [Электронный ресурс]. URL: https://learn.microsoft.com/en-us/cpp/assembler/inline/inline-assembler?view=msvc-170 (дата обращения: 03.10.2025).
4. Inline Assembler. D programming language specification. [Электронный ресурс]. URL: https://dlang.org/spec/iasm.html (дата обращения: 03.10.2025).
5. Rust RFC-2873: inline assembly [Электронный ресурс]. URL: https://rust-lang.github.io/rfcs/2873-inline-asm.html (дата обращения: 03.10.2025).
6. Linux Kernel Community. KVM: Kernel-based Virtual Machine [Электронный ресурс]. 2007. URL: https://www.linux-kvm.org/page/Main_Page (дата обращения: 07.10.2026).
7. Microsoft Corporation. Hyper-V: Deploying and Managing Virtual Machines [Электронный ресурс]. 2023. URL: https://learn.

microsoft.com/en-us/virtualization/hyper-v-on-windows/about/ (дата обращения: 07.10.2025).

8. Recoules F. и др. Get Rid of Inline Assembly through Verification-Oriented Lifting // 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2019. Сс. 577–589.
9. Shadrin A. Mixed low- and high level programming language semantics and automated verification of a small hypervisor. 2012.
10. Patterson D. и др. FunTAL: reasonably mixing a functional language with assembly // Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. Barcelona, Spain: Association for Computing Machinery, 2017. Сс. 495–509.
11. GCC GNU Compiler Collection Documentation: Inline Assembly, C Extensions and RTL Representation [Электронный ресурс]. URL: https://gcc.gnu.org/onlinedocs/gccint/ (дата обращения: 13.10.2025).
12. Merrill J. Generic and gimple: A new tree representation for entire functions. 2003.
13. Clang: a C language family frontend for LLVM [Электронный ресурс]. URL: https://clang.llvm.org/ (дата обращения: 13.10.2025).
14. LLVM Compiler Infrastructure Documentation [Электронный ресурс]. URL: https://llvm.org/ (дата обращения: 13.10.2025).
15. Sharma M. LLVM Techniques, Tips, and Best Practices: Clang and Middle-End Libraries: Design Powerful and Reliable Compilers Using the Latest Libraries and Tools from LLVM. Birmingham: Packt Publishing, 2022. С. 330.

16. LLVM Compiler Infrastructure Documentation: Bitcode Format [Электронный ресурс]. URL: https://llvm.org/docs/BitCodeFormat.html (дата обращения: 01.11.2025).
17. The Java HotSpot Performance Engine Architecture [Электронный ресурс]. URL: https://www.oracle.com/java/technologies/whitepaper.html (дата обращения: 06.04.2026).
18. The Graal Virtual Machine [Электронный ресурс]. URL: https://www.graalvm.org/ (дата обращения: 06.04.2026).
19. Recoules F. и др. RUSTInA: Automatically Checking and Patching Inline Assembly Interface Compliance (Artifact Evaluation)" // 2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). 2021. Сс. 201–202.
20. Djoudi A., Bardin S. BINSEC: Binary Code Analysis with Low-Level Regions // Tools and Algorithms for the Construction and Analysis of Systems / под ред. Baier C., Tinelli C. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015. Сс. 212–217.
21. Click C. Global Code Motion / Global Value Numbering. 1995. Т. 30. Сс. 246–257.

ПРИЛОЖЕНИЕ А

Листинги объединенных реализаций вставок

Данные листинги представляют переписанную версию двух реализаций ассемблерной вставки *CoroutineUtils* для архитектур *aarch64*, *x86-64*. Реализации имеют схожую структуру: работа с указателем на стек, сохранение регистров общего назначения и вещественнозначных регистров, работа с указателем на стек, восстановление тех же регистров.

```
; sp_offs, gprs_offs, xmms_offs - вычисляются во время компиляции

JR_ControlTransfer:
  ; this == rcx
  ; another == rsi

  ; сохранение регистра указателя на стек
  mov [rcx, sp_offs], rsp

  ; сохранение для rbx, rbp, r11, r12, r13, r14, r15
  mov [rcx, gprs_offs + 0*0x8], rbx
  mov [rcx, gprs_offs + 1*0x8], rbp
  ...

  ; сохранение xmm6, xmm7, xmm10, xmm11, xmm12, xmm13, xmm14, xmm15
  movdqu [rcx + xmms_offs + 0*0x10], xmm6
  ...

  ; восстановление регистра указателя на стек
  mov rsp, [rsi + sp_offs]

  mov rbx, [rsi + gpr_offs + 0*0x8]
  ...

  movdqu xmm6, [rsi + xmms_offs + 0*0x8]
  ...

  ret
```

Листинг А.22 — coroutine-utils x86-64

```
; sp_offs, gprs_offs, vreg_offs - вычисляются во время компиляции

JR_ControlTransfer:
  mov x4, sp
  stp x4, lr, [x0, sp_offs]

  ; сохранение пар регистров (x19 x20), (x21 x22), (x23 x24), (x25 x26), (x28 x29)
  stp x19, x20, [x0, gprs_offs + 0*16]
  ...
  str x27, [x0, gpr_offs + 5*16]

  ; сохранение пар регистров (q10 q11) (q12 q13) (q14 q15)
  stp q8, q9 [x0, vreg_offs + 0*32]
  ...

  ldp x4, lr, [x1, sp_offs]
  mov sp, x4

  ; ниже - восстановление
  ldp x19, x20, [x1, gprs_offs + 0 * 16]
  ...

  ldp q8, q9, [x1, vregs_offs + 0 * 32]
  ...

  ret
```

Листинг A.23 — coroutine-utils aarch64

Поэтому обобщенная версия выглядит следующим образом:

а) вводятся методы *coroutineUtilsPrologue*, *coroutineUtilsEpilogue*, которые выполняют работу по сохранению и восстановлению регистра указателя на стек.

б) метод *OP_REGS* в цикле вызывает переопределенные методы *STORE_GPRS_PAIR*, *STORE_GPR*, *STORE_FREGS_PAIR*, *LOAD_GPR_PAIR*, *LOAD_GPR*, *LOAD_FREG_PAIR* (в случае *x86-64* методы работающие с парами регистров не

переопределены), сохраняя/восстанавливая регистры нужного типа.

в) Сохраняемые регистры выделены в обобщенные множества, которые возвращают методы *GPR_REGS*, *FREGS* (переопределенные для архитектур).

г) Константы вроде *SP_OFFS* вычисляются при компиляции.

```
private def gen(filled: IRegister, loaded: IRegister): Unit = {
  emit.coroutineUtilsPrologue(filled, SP_OFFS.x)

  // store
  OP_REGS(filled, STORE_GPRS_PAIR, STORE_GPR, STORE_FREGS_PAIR)

  emit.coroutineUtilsEpilogue(loaded, SP_OFFS.x)

  // load
  OP_REGS(loaded, LOAD_GPR_PAIR, LOAD_GPR, LOAD_FREG_PAIR)
}
```

Листинг А.24 — обобщенное тело вставки CoroutineUtils для архитектур aarch64, x86-64

ПРИЛОЖЕНИЕ Б

Листинги сгенерированного кода

В приложении приведены фрагменты сгенерированного машинного кода для реализации ассемблерных вставок через вызов внешнего метода и через специализированные вершины промежуточного представления. Листинги использовались для качественного анализа изменений в структуре кода, включая сокращение числа межрегистровых пересылок и устранение инструкций сохранения и восстановления регистров.

```
...
mov ebp, eax
mov eax, 0x0
lock cmpxchg dword ptr [r8], r9d
mov eax, ebp
...
```

Листинг Б.25 — сгенерированный код CAS для бенчмарка Register pressure с использованием интерфейса

```
...
mov r12,rcx
mov rcx, rax
mov r13, rsi
mov esi, 0x0
mov r11, rdx
mov edx, edi
call 8306d40 ; вызов CAS
mov rcx, r12
mov rsi, r13
mov rdx, r11
...
```

Листинг Б.26 — сгенерированный код CAS для бенчмарка Register pressure в виде внешнего метода

```
mov ecx, 0x0
jmp 8308831
lea rdx, [rip+0x2fb78a]
mov edx, dword ptr [rdx+rcx*4]
mov eax, edx
bswap rax
lea rsi, [rip+0xc1b797c]
mov esi, dword ptr [rsi+rcx*4]
imul edx, esi
add edx, eax
lea rax, [rip+0x23f2fd6d]
mov dword ptr [rax+rcx*4], edx
add ecx, 0x1
cmp ecx, 0x2faf080
jl 8308807
ret
```

Листинг Б.27 — сгенерированный код bitswap для бенчмарка Register pressure с использованием интерфейса

```
push rbp
push rbx
add rsp, 0xfffffffffffffff8
mov ecx, 0x0
jmp 8308881
lea rdx, [rip+0x2fb7c4]
mov edx, dword ptr [rdx+rcx*4]
mov eax, edx
lea rsi, [rip+0xc1b79b8]
mov esi, dword ptr [rsi+rcx*4]
imul edx, esi
mov ebx, esi
mov ecx, eax
mov ebp, edx
call 82e7df0
add ecx, ebp
lea rbp, [rip+0x23f2fd9e]
mov dword ptr [rbp+rbx*4], ecx
add ecx, [rbx+0x1]
cmp ecx, 0x2faf080
jl 830884d
add rsp, 0x8
pop rbx
pop rbp
ret
```

Листинг Б.28 — сгенерированный код bitswap для бенчмарка Register pressure в виде внешнего метода