Современные виртуальные машины

Кирилл Батузов

ИСП РАН

23 октября 2013

Виртуальные машины

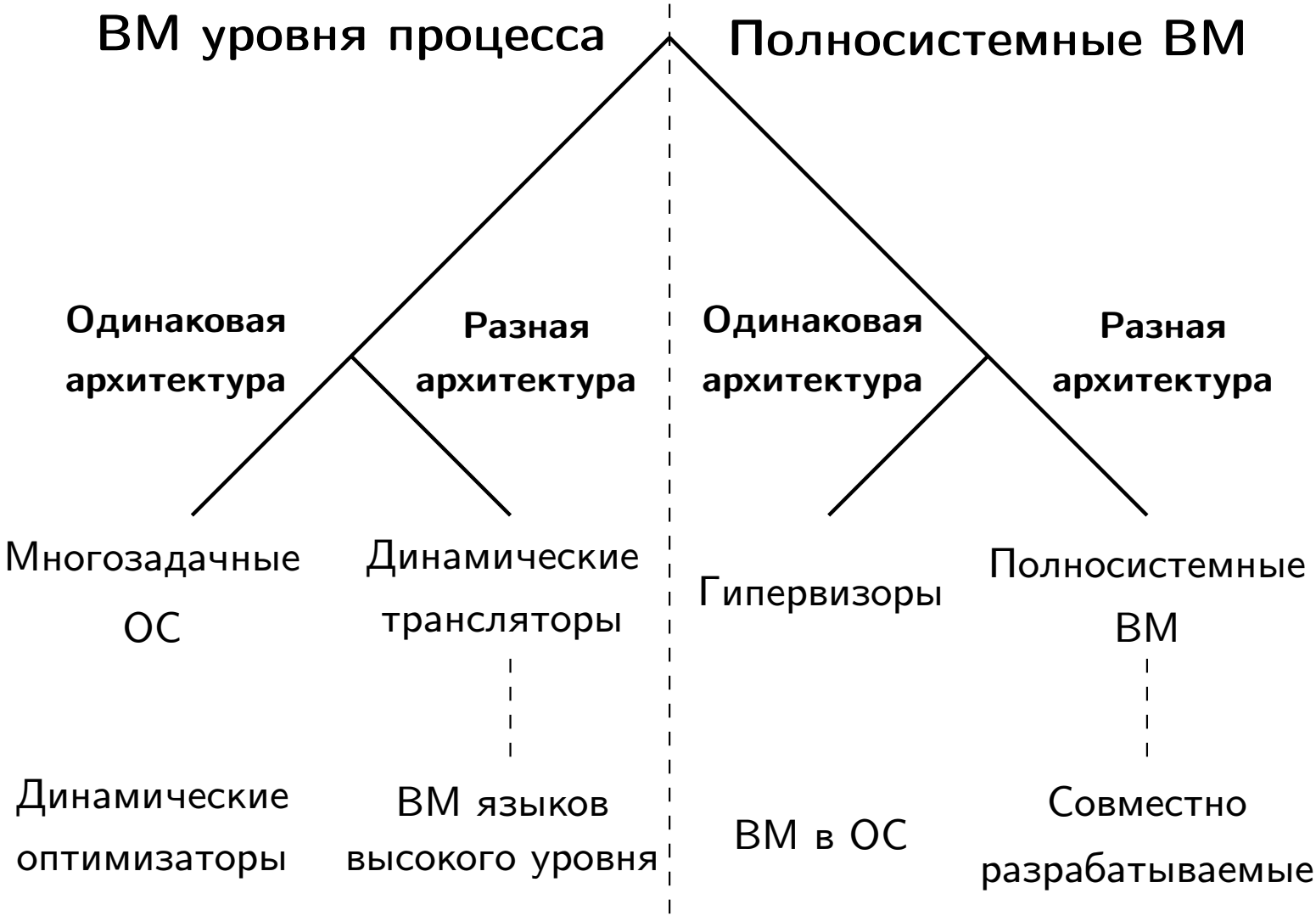
Эмуляция — это процесс реализации интерфейса и функциональности одной системы или подсистемы поверх другой, имеющей отличный интерфейс. Полученная система или подсистема при этом называется виртуальной.

Виртуальные машины — частный случай таких эмулируемых систем. Существуют различные типы виртуальных машин в зависимости от того, какой интерфейс эмулируется (ISA или ABI).

Применение виртуальных машин

- Консолидация серверов и облачные вычисления.
- Создание набора различных окружений. Виртуальное аппаратное обеспечение.
- Тестирование и обеспечение качества.
- Отладка.
- Контролируемое окружение и его применение в задачах компьютерной безопасности.

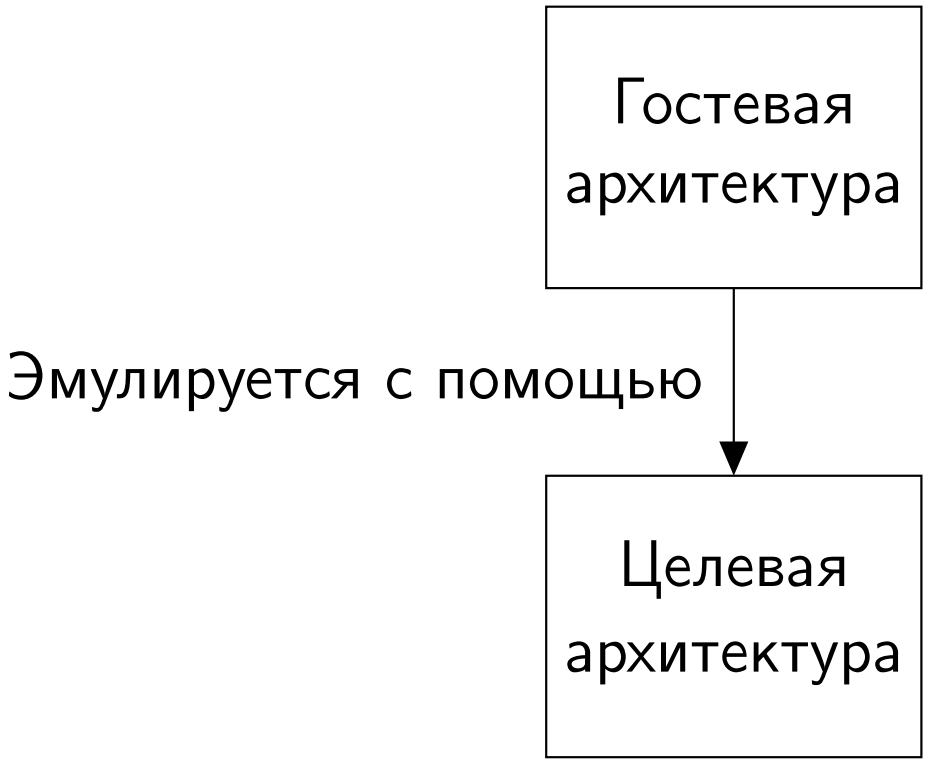
Классификация виртуальных машин



Эмуляция системы команд

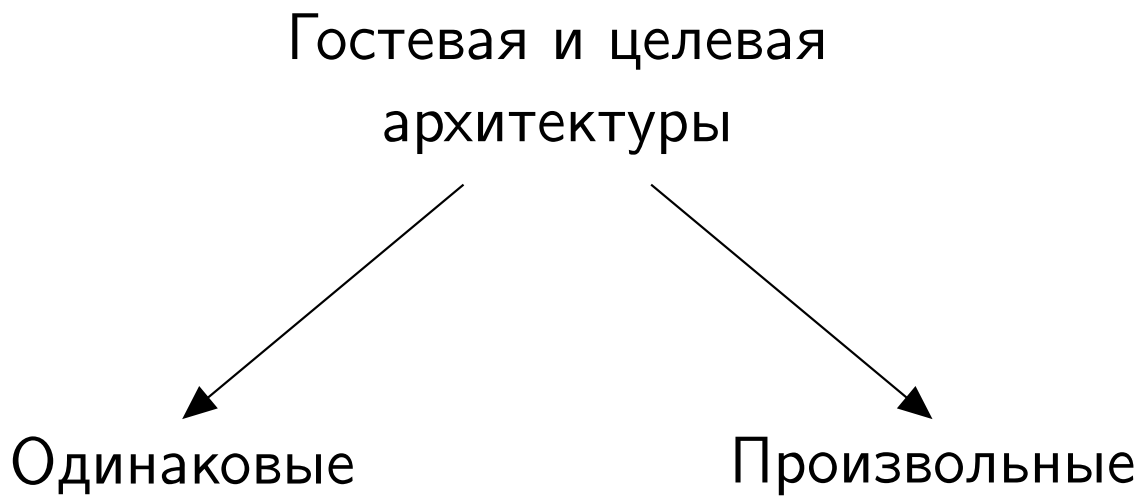
Эмуляция системы команд

Различным типам виртуальных машин так или иначе приходится решать одну общую задачу: эмуляцию системы команд. Эмулируемую систему команд будем называть гостевой, архитектуру же на которой выполняется код в реальности будем называть целевой.



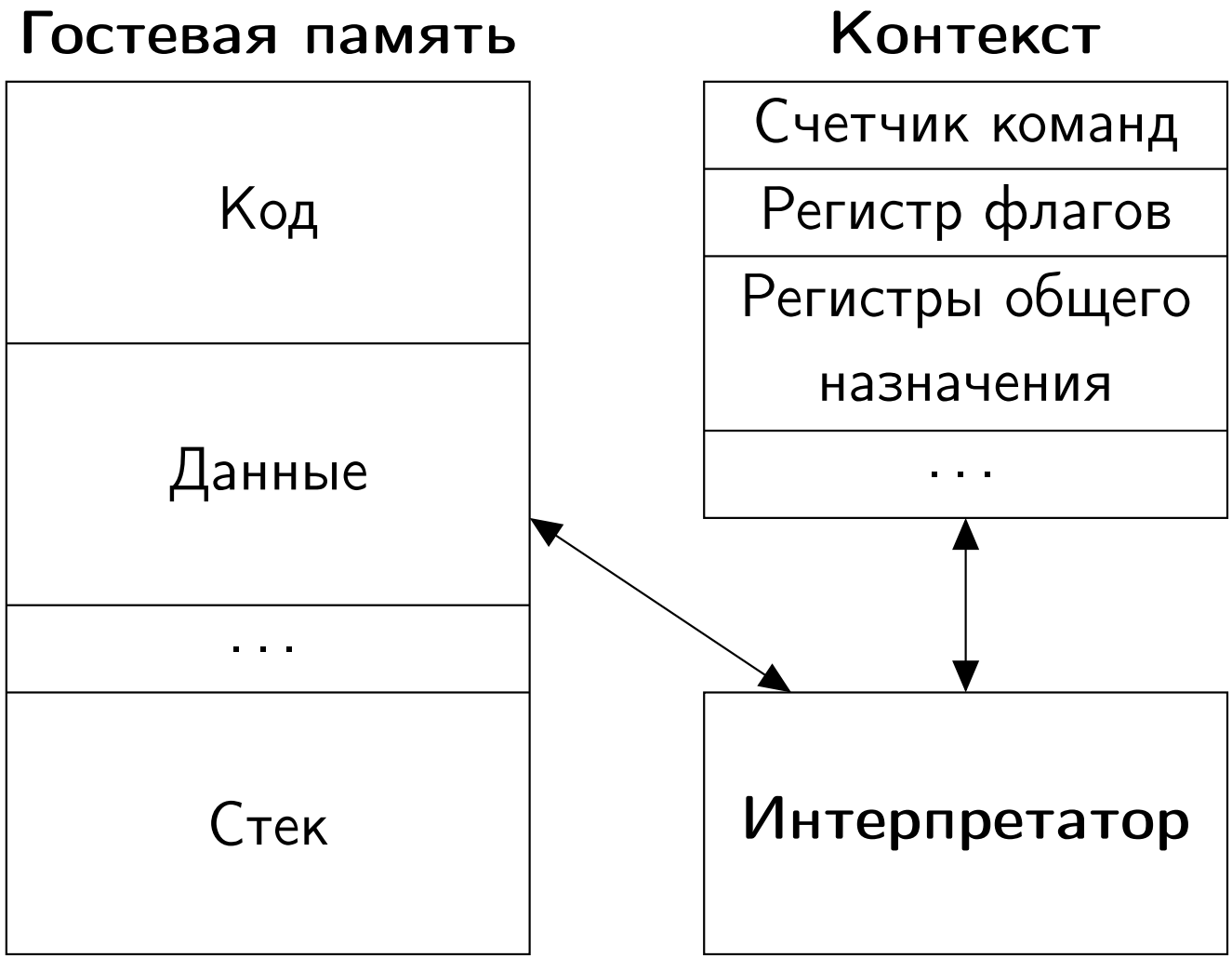
Эмуляция системы команд

Есть два существенно различающихся случая.



Рассмотрим сначала общий случай, а затем разберем случай одинаковых архитектур.

Простая интерпретация



Простая интерпретация

Простой интерпретатор инструкций PowerPC

```
while (!halt && !interrupt) {
    inst = code[PC];
    opcode = extract(inst, 31, 6);
    switch (opcode) {
        case LoadWordAndZero: LoadWordAndZero(inst);
        case ALU: ALU(inst);
        case Branch: Branch(inst);
        ...
    }
}
```

Последовательная интерпретация

Оптимизация: можно сократить число переходов, если в конец каждой функции, обрабатывающей отдельные типы инструкций, добавить код основного цикла.

Прямой переход на следующую инструкцию

```
void LoadWordAndZero(inst) {
    ...
    if (halt || interrupt) return ;
    inst = code[PC];
    opcode = extract(inst, 31, 6);
    extended_opcode = extract(inst, 10, 10);
    routine = dispatch[opcode][extended_opcode];
    routine(inst);
}
```

Примечание: будем считать что компилятор умеет разворачивать хвостовую рекурсию.

Предекодирование

Оптимизация: можно сократить количество битовых операций **extract** предварительно декодировав все инструкции в более удобное для интерпретатора представление.

Удобное представление инструкции

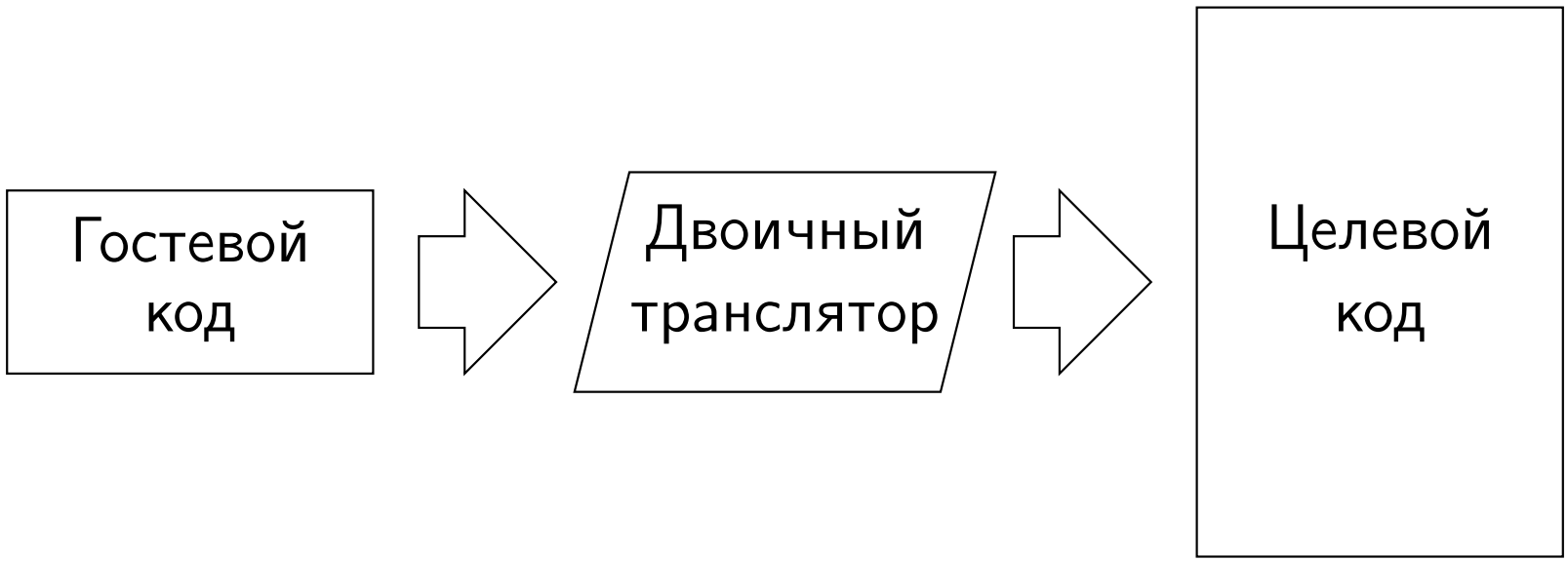
```
struct instruction {
    unsigned long op;
    unsigned char dest;
    unsigned char src1;
    unsigned int src2;
} code[CODE_SIZE];
```

Оптимизация: вместо кода операции можно сразу вычислять и записывать адрес обрабатывающей функции эмулятора.

Двоичная трансляция

Определение

Процесс преобразования программы из одного двоичного представления в другое называется двоичной трансляцией.

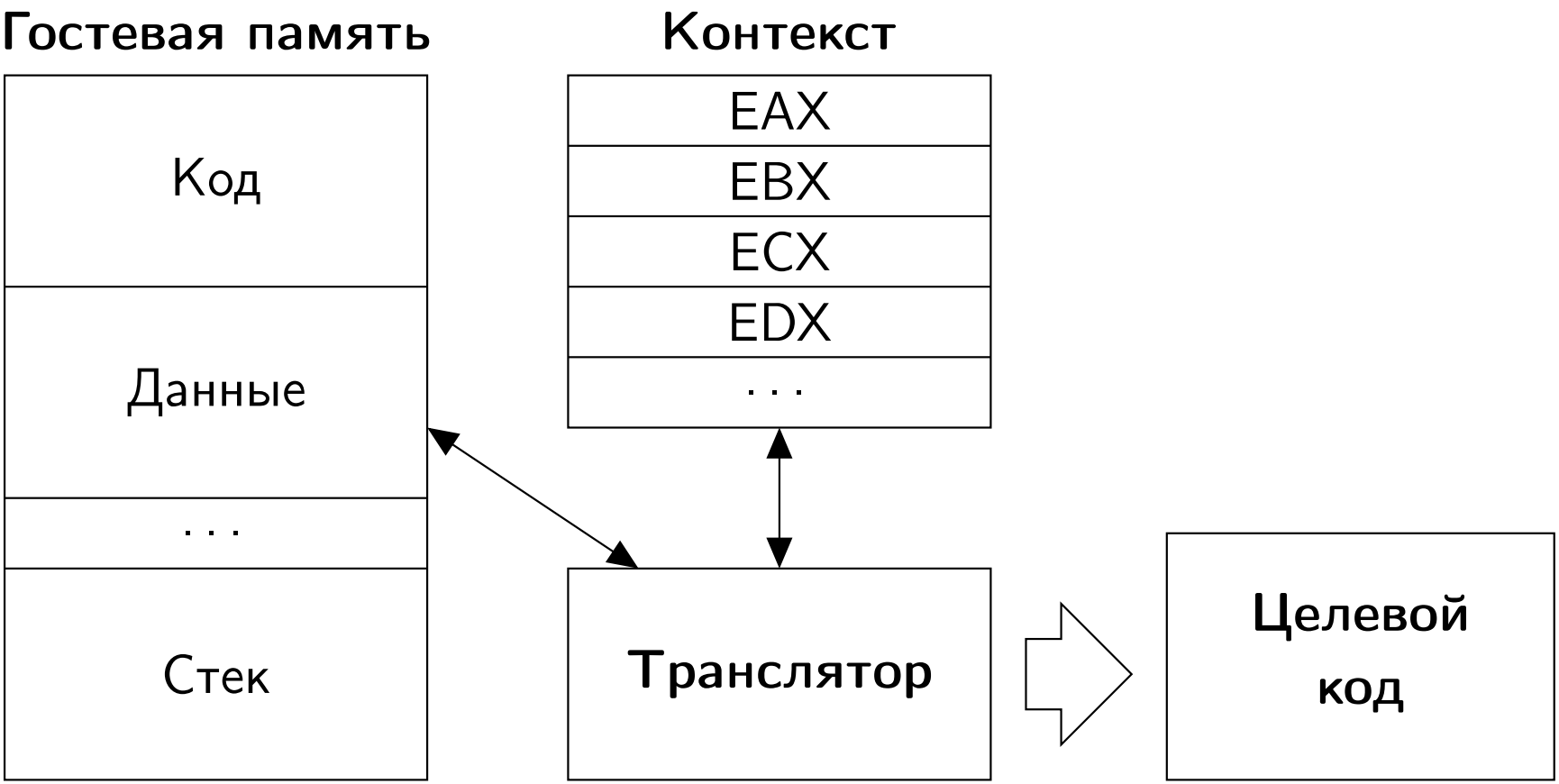


Пример трансляции x86 → PowerPC

r1 содержит указатель на контекст

r2 содержит указатель на начало гостевой памяти

r3 содержит гостевой счетчик команд



Пример трансляции x86 → PowerPC

ADD EDX, dword ptr [EAX + 4]

```
lwz     r4, 0(r1)     ; r4  <- EAX
addi    r5, r4, 4     ; r5  <- EAX + 4
lwzx    r5, r2, r5    ; r5  <- [EAX + 4]
lwz     r4, 12(r1)    ; r4  <- EDX
add     r5, r4, r5    ; r5  <- [EAX + 4] + EDX
stw     r5, 12(r1)    ; EDX <- r5
addi    r3, r3, 3     ; EIP += 3
```

ADD EAX, 4

```
lwz     r4, 0(r1)     ; r4  <- EAX
addi    r4, r4, 4     ; r4  <- EAX + 4
stw     r4, 0(r1)     ; EAX <- r4
addi    r3, r3, 3     ; EIP += 3
```

Двоичная трансляция: проблемы

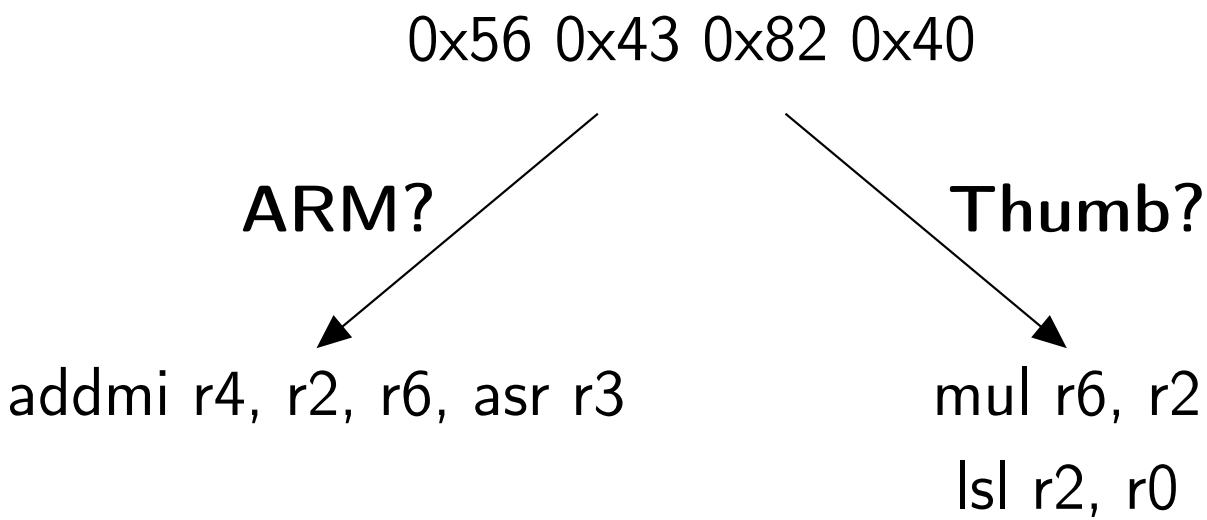
- Проблема обнаружения кода.
  - В архитектуре фон Неймана невозможно отличить данные и код.
  - Нередко один и тот же фрагмент памяти может декодироваться в различные инструкции в зависимости от контекста.

Двоичная трансляция: проблемы

- Декодирование инструкций x86

			mov CH, 0 ?
31	c0	8b	b5 00 00 03 08 8b bd 00 00 03 00
			mov ESI, [EBP + 0x08030000] ?

- Декодирование инструкций ARM



Двоичная трансляция: проблемы

- Проблема обнаружения кода.
  - В архитектуре фон Неймана невозможно отличить данные и код.
  - Нередко один и тот же фрагмент памяти может декодироваться в различные инструкции в зависимости от контекста.
- Проблема нахождения кода.
  - В случае перехода по адресу, записанному в регистре, оказывается вычислен гостевой адрес. Вычисление соответствующего ему целевого адреса является нетривиальной задачей.

JMP EBX

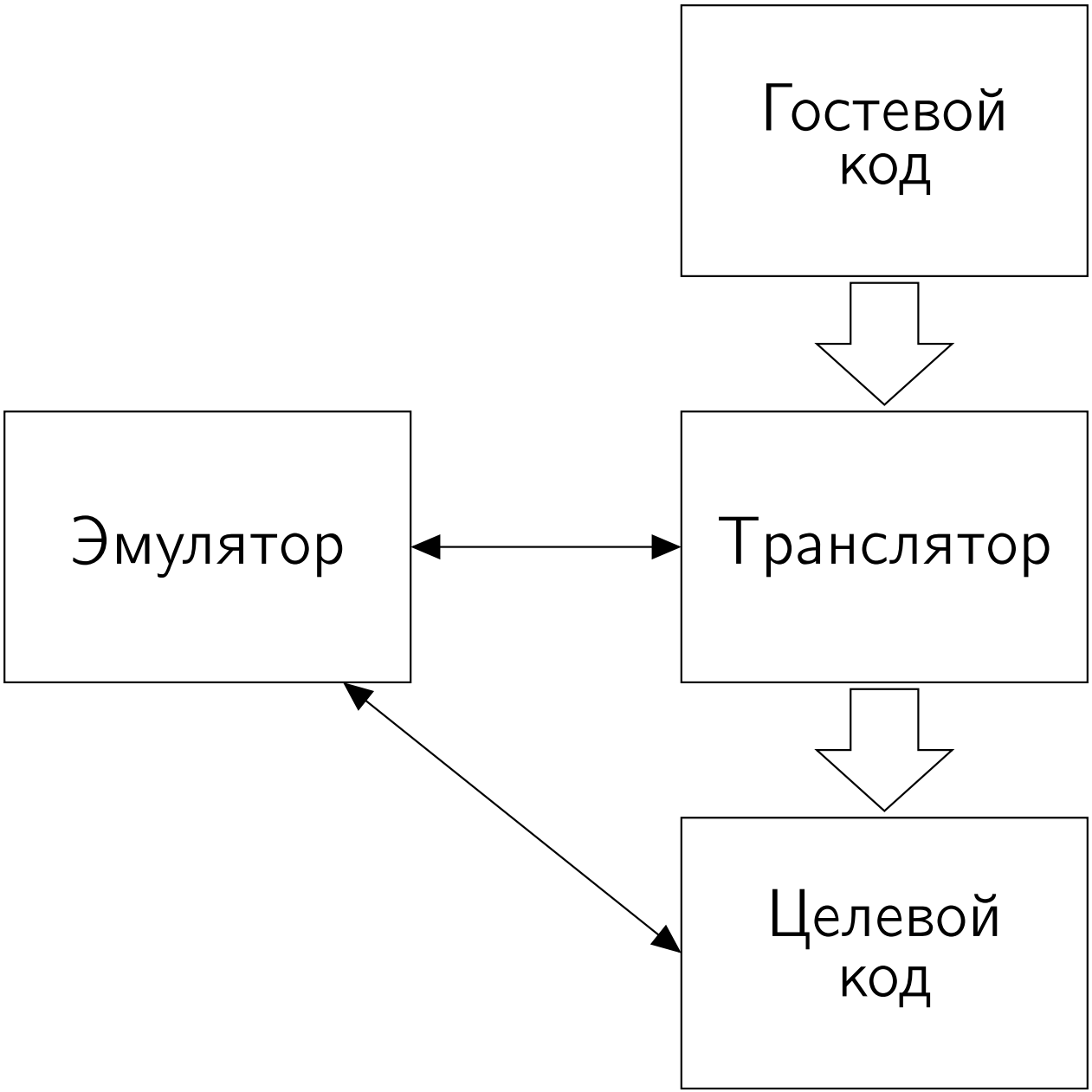
```
lwz     r4, 4(r1)          ; r4  <- EBX
mtctr   r4                 ; ctr <- EBX
bctr                       ; Jump ctr. Ooops...
```

Двоичная трансляция: проблемы

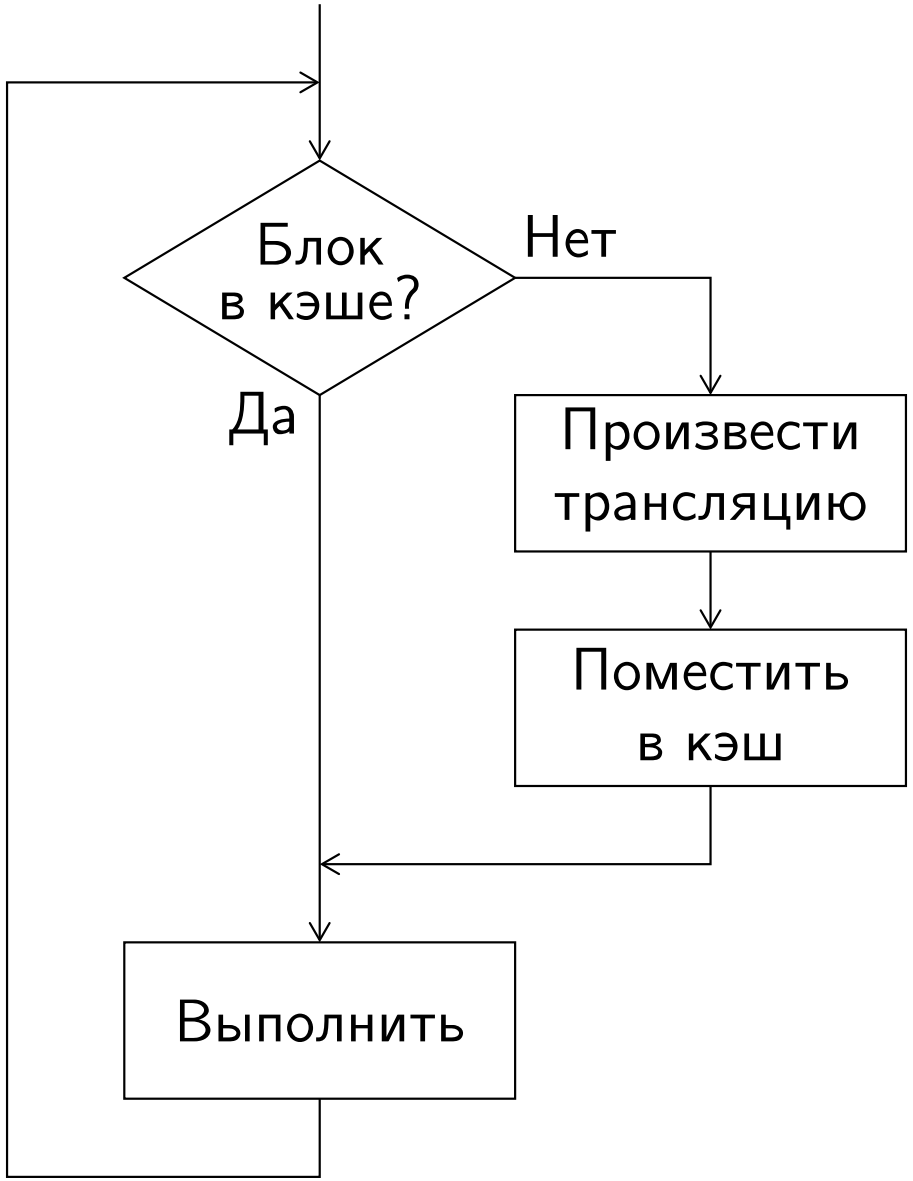
- Проблема обнаружения кода.
  - В архитектуре фон Неймана невозможно отличить данные и код.
  - Нередко один и тот же фрагмент памяти может декодироваться в различные инструкции в зависимости от контекста.
- Проблема нахождения кода.
  - В случае перехода по адресу, записанному в регистре, оказывается вычислен гостевой адрес. Вычисление соответствующего ему целевого адреса является нетривиальной задачей.
- Проблема самомодифицирующегося кода.
  - Код программы может изменяться в ходе ее выполнения.

Инкрементальная (динамическая) трансляция

- Код транслируется во время выполнения при первом обращении.
- Трансляция производится небольшими фрагментами кода, которые называют блоками трансляции.
- Полученные трансляции помещаются в кэш и используются при повторных обращениях.
- Кэш имеет механизмы:
  - быстрого поиска блоков в нем,
  - вытеснения старых блоков при переполнении,
  - удаления «испорченных» блоков в случае модификации кода.


Гостевой код
Эмулятор
Транслятор
Целевой код

Динамическая трансляция: блок-схема



Сцепление блоков трансляции

- После выполнения блока гостевого кода управление передается эмулятору, который находит следующий блок для выполнения и передает управление ему.
- Если в конце блока A переход выполняется по фиксированному адресу, то при каждом таком вычислении будет находиться один и тот же блок B.
- Блок A можно модифицировать так, чтобы он сразу передавал управление блоку B минуя эмулятор. Но! Сделать это возможно только после того, как оба блока оттранслированы и находятся в кэше.
- Необходимо предусмотреть возможность разрыва таких соединений, чтобы вернуть управление эмулятору для обработки событий (прерываний).

Ленивое вычисление флагов

- Вычисление флагов — сложная медленная операция, которая занимает больше времени, чем непосредственная эмуляция инструкции.
- В некоторых архитектурах (например, x86) большинство инструкций устанавливают флаги.
- Инструкций, использующих флаги довольно мало и встречаются они относительно редко.
- Оптимизация: вместо вычисления флагов можно просто запоминать последнюю операцию, которая их изменила, и ее аргументы. Флаги вычисляются только в момент их использования.

Сравнение

	Использование памяти	Начальная производительность	Итоговая производительность	Переносимость
Интерпретация	Низкое	Высокая	Низкая	Хорошая
Предекодирование	Высокое	Низкая	Средняя	Средняя
Двоичная трансляция	Высокое	Низкая	Высокая	Плохая

Случай одинаковых архитектур

В случае, когда гостевая и основная архитектуры совпадают, возникает желание избежать трансляции, и просто выполнять код напрямую. Однако необходимо обеспечивать следующие свойства.

- Любая программа, выполняющаяся под управлением виртуальной машины, должна иметь идентичное поведение.
- Виртуальная машина должна сохранять полный контроль над реальными ресурсами системы.

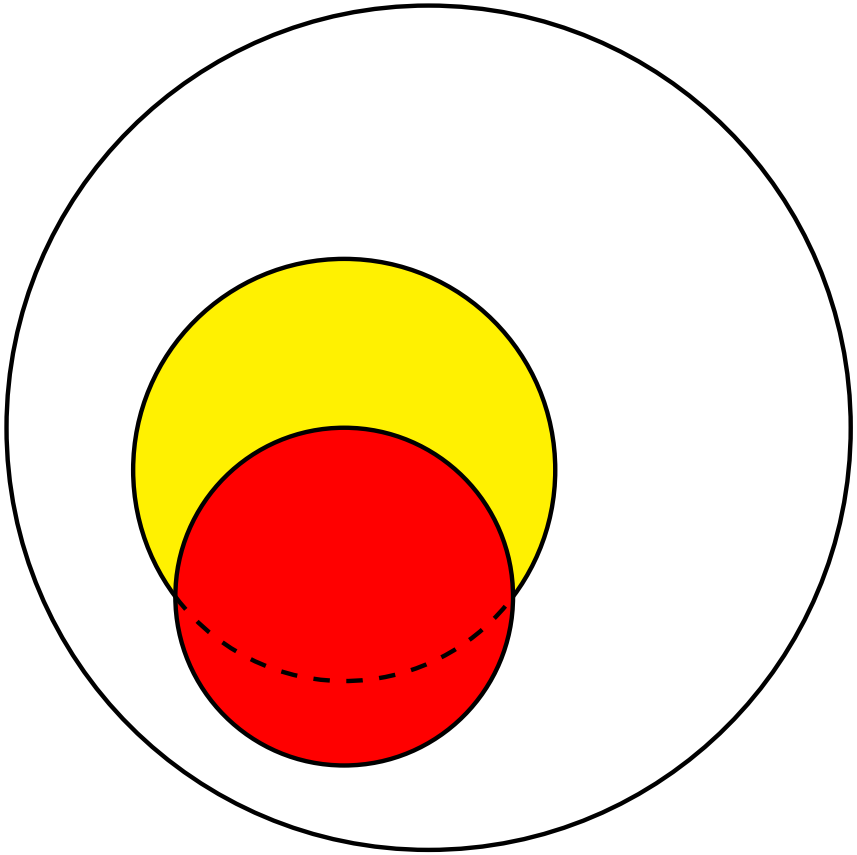
Привилегированные и чувствительные инструкции

- Инструкция называется привилегированной, если она может быть выполнена только в привилегированном режиме, а попытка выполнить ее в непривилегированном режиме приводит к исключению.
  - Пример: CLI и STI архитектуры x86.
- Инструкция называется чувствительной, если она пытается управлять ресурсами системы или ее результат зависит от режима выполнения.
  - Пример: POPF архитектуры x86.

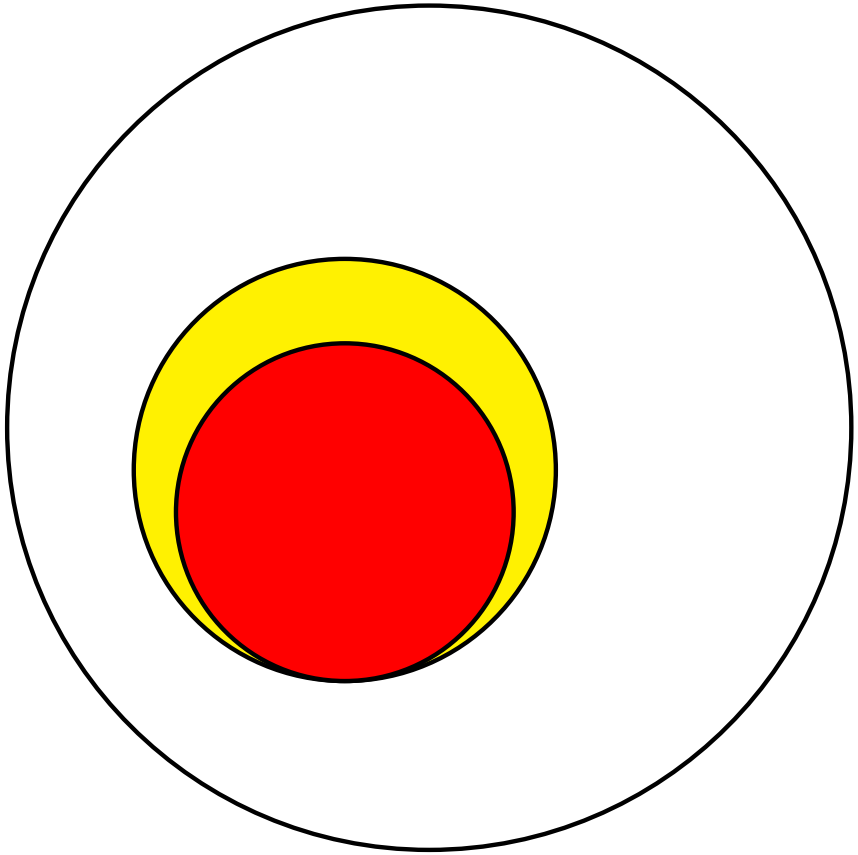
Теорема Попека-Гольдберга

Построение эффективной виртуальной машины возможно, если множество чувствительных инструкций является подмножеством привилегированных.

Плохо виртуализуется



Хорошо виртуализуется



Все инструкции
Привилегированные
Чувствительные

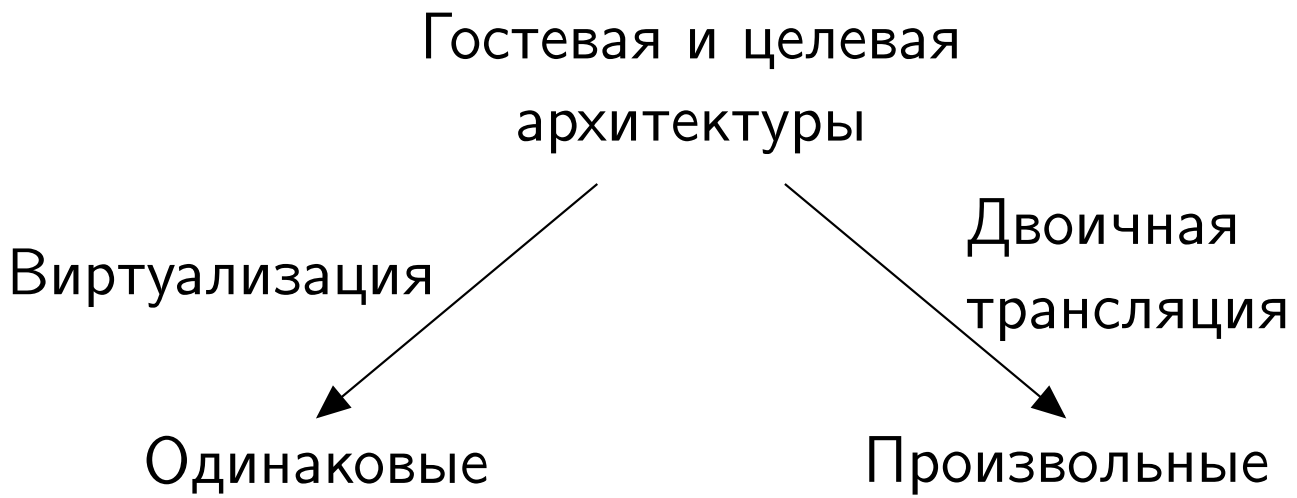
Построение эффективной виртуальной машины

- В случае выполнения условий теоремы Попека-Гольдберга
  - Виртуальная машина работает в привилегированном режиме.
  - Весь гостевой код выполняется в непривилегированном режиме.
  - Привилегированные инструкции перехватываются и эмулируются виртуальной машиной.
- Иначе...
  - Построение эффективной виртуальной машины все равно возможно, однако сделать это значительно сложнее.
  - Производительность все равно будет ниже.
  - Можно, например, сканировать код перед выполнением и заменять «плохие» инструкции на явное прерывание.

Подведем итог

Эмуляция системы команд

Были рассмотрены два существенно различающихся случая.



Литература

James E. Smith, Ravi Nair. “Virtual Machines: Versatile Platforms for Systems and Processes”