УДК 004.45

ИССЛЕДОВАНИЕ ПРИМЕНИМОСТИ ВИРТУАЛЬНЫХ МАШИН В ЖИЗНЕННОМ ЦИКЛЕ ПРОМЫШЛЕННОГО ИНТЕРНЕТА ВЕЩЕЙ

Фатнев Р.А., Казанцева Л.В., Тарасов И.Е.

МИРЭА - Российский технологический университет, 119454, Россия, г. Москва, проспект Вернадского, 78,
e-mail: rfatnev@gmail.com, larakazantseva@gmail.com, tarasov_i@mirea.ru

Существует несколько стандартов, описывающих архитектуру промышленного интернета вещей, но в общем случае любая архитектура IIoT включает в себя уровень устройства, обеспечивающий функциональность устройств и шлюзов IIoT. В данной статье исследуется возможность применения виртуальных машин для решения некоторых проблем реализации уровня устройства сети IIoT. Главными задачами выступают исследование проблем внедрения виртуальных машин в жизненный цикл IIoT и описание возможных решений.

Ключевые слова: виртуальная машина, языково-ориентированное программирование, предметно-ориентированный язык, промышленный интернет вещей, архитектура промышленного интернета вещей

RESEARCH OF THE APPLICABILITY OF VIRTUAL MACHINES IN THE LIFE CYCLE OF THE INDUSTRIAL INTERNET OF THINGS

Fatnev R.A., Kazantceva L.V., Tarasov I.E.

MIREA - Russian Technological University, 119454, Moscow, 78 Vernadskogo Avenue, Russia
e-mail: rfatnev@gmail.com, larakazantseva@gmail.com, tarasov_i@mirea.ru

There are several standards that describe the architecture of the Industrial Internet of Things, but in general, any IIoT architecture includes a device layer that provides the functionality of IIoT devices and gateways. This article explores the possibility of using virtual machines to solve some of problems of implementing the device layer of an IIoT network. The main tasks are to research the problems of integrating virtual machines into IIoT lifecycle and describe possible solutions.

Keywords: virtual machine, language-oriented programming, domain specific language, industrial internet of things, architecture of industrial internet of things

Введение

Промышленный интернет вещей - это система, состоящая из объединенных в сеть интеллектуальных объектов, киберфизических активов, связанных с ними общих информационных технологий и дополнительных облачных или периферийных вычислительных платформ, которые обеспечивают интеллектуальный и автономный доступ в реальном времени, сбор, анализ, обмен данными и обмен процессами, продуктами и / или служебной информацией в промышленной среде, чтобы оптимизировать общую стоимость производства [3]. Существует несколько стандартов, описывающих архитектуру промышленного интернета вещей. Согласно публикации Промышленного Интернет Консорциума [6] референсная архитектура интернета вещей служит шаблоном для разработки различных систем IoT (IIoT в частности). Кроме того, ISO (Международная организация по стандартизации) и IEC (Международная электротехническая комиссия) создали технический комитет ISO/IEC JTC 1 в области информационных технологий и сформировали собственный стандарт ISO/IEC 21823-1 IoT [7], описывающий архитектуру интернета вещей. В общем случае, любая архитектура промышленного интернета вещей является частью эталонной модели интернета вещей, описанной в рекомендации ITU-T Y.4000/Y2060 (рис. 1).

Эта модель включает в себя 4 основных уровня: устройства, сети, поддержки и приложения. Также выделены 2 дополнительных уровня: управления и безопасности. Уровень устройства обеспечивает функциональность устройств и шлюзов. Уровень сети обеспечивает организацию устройств в единую сеть и транспортировку данных внутри этой сети. Уровень поддержки обеспечивает удовлетворение требований различных приложений IoT, например, сбор, обработка и хранение данных. Уровень приложения является верхним уровнем модели и

включает в себя приложения IoT. Уровни управления и безопасности являются сквозными и обеспечивают функции, специфичные для домена, и защиту данных на всех уровнях.

Рисунок 1. Эталонная модель IoT по рекомендации ITU-T Y.4000/Y2060.

Любая архитектура IIoT содержит уровень устройства, обеспечивающий функциональность устройств и шлюзов в сети. Существует большое множество различных устройств для различных функций, которые, зачастую, имеют разную аппаратную платформу и программируются с учетом их специфики, что не позволяет переиспользовать существующее программное обеспечение на других платформах. Платформозависимое ПО часто препятствует масштабированию сети (как горизонтальному, так и вертикальному). Если при масштабировании появляется необходимость миграции платформозависимого ПО на другую аппаратную платформу, разработчики вынуждены адаптировать большую часть ПО под специфику новой платформы, что влечет большие финансовые и временные затраты. Также платформозависимое ПО усложняет процесс обеспечения интероперабельности устройств с разной аппаратной платформой из-за необходимости учета всех их специфик при организации сети.

Также проблемой разработки уровня устройств IIoT является ограниченность ресурсов. Устройства IIoT часто обладают объемом памяти, не превышающим несколько мегабайт (а иногда и несколько десятков килобайт). При росте сложности системы объем программного кода увеличивается, как и затраты памяти устройств. Таким образом, при расширении системы, разработчики могут столкнуться с нехваткой памяти для хранения программного обеспечения устройств IIoT.

Уровень виртуализации устройств

Для решения проблемы платформозависимого программного обеспечения необходимо абстрагироваться от специфик аппаратных платформ различных устройств. Сделать это можно путем добавления дополнительного уровня в модель - уровня виртуализации устройств (рис. 2).

Уровень виртуализации устройств должен предоставлять виртуальные функции устройства, не зависящие от аппаратной платформы. При использовании виртуализации, необходимость учета специфики конкретных физических устройств отпадает, поскольку программное обеспечение будет писаться под виртуальные устройства. Такой подход позволит писать кроссплатформенное программное обеспечение. Кроссплатформенность — это свойство программного обеспечения, позволяющее ему работать с несколькими аппаратными платформами.

Одним из способов реализации уровня виртуализации устройств является подход применения виртуальных машин, в частности сред языков программирования. Использование виртуальных машин позволяет абстрагироваться от аппаратной платформы. Виртуальные машины имеют собственный набор команд, который не зависит от платформы, под которую написана виртуальная машина. Тем самым, при разработке сети IIoT программное обеспечение для устройств пишется под виртуальную платформу, аппаратная часть которой может быть заменена, не затрагивая само программное обеспечение.

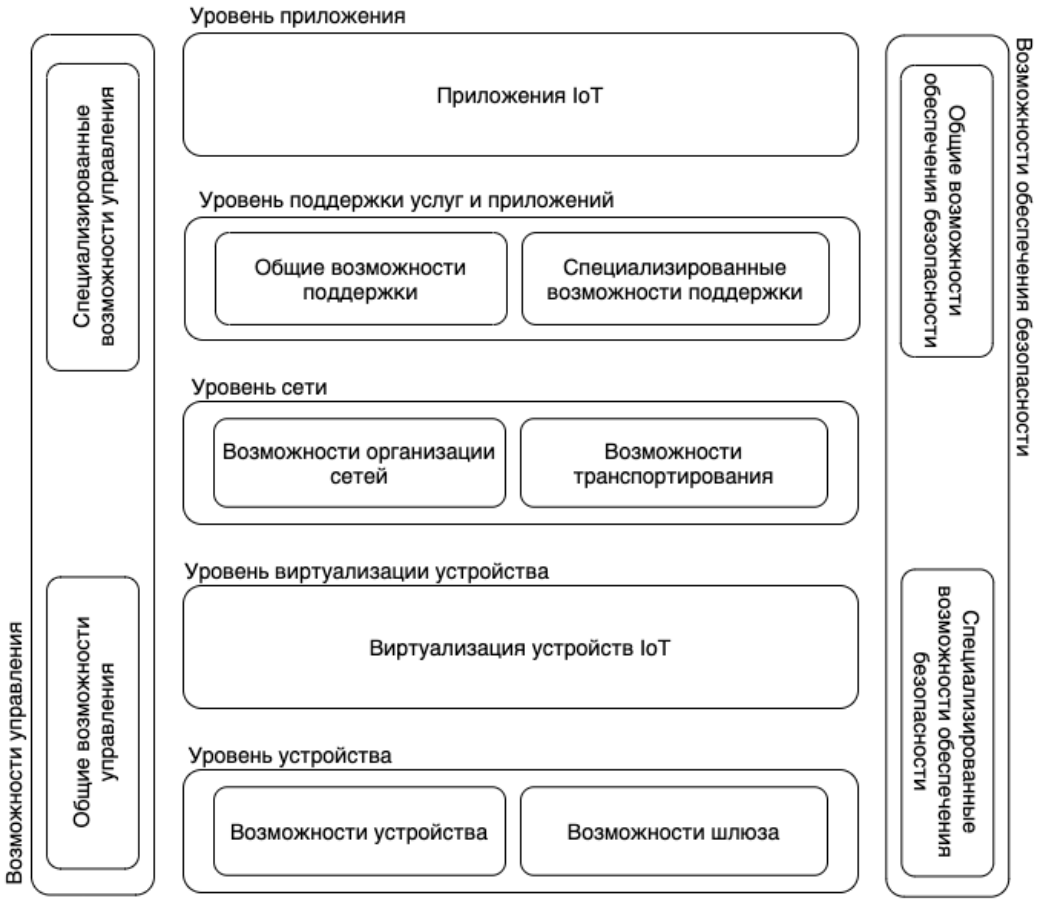

Рисунок 2. Уровень виртуализации устройств в модели IoT по рекомендации ITU-T Y.4000/Y2060.

На данный момент существует множество виртуальных машин для устройств с ограниченными ресурсами. Примеры таких ВМ описаны в таблице 1. Но программирование под такие ВМ осуществляется при помощи языков общего назначения, и они часто не подходят для применения подхода языково-ориентированного программирования. Языково-ориентированное программирование — это парадигма программирования, заключающаяся в разбиении процесса разработки на стадии разработки предметно-ориентированного языка (DSL - domain specific language) и описания решения задач с его использованием. Реализация специализированных виртуальных машин под предметную область может облегчить применение подхода языково-ориентированного программирования путем предоставления команд как низкого уровня абстракции (например, сохранение значения в памяти, математические операции), так и высокого (например, передача значений по UART), что облегчает компиляцию исходного кода DSL в байт-код виртуальной машины. Кроме того, виртуальная машина с наличием команд высокого уровня абстракции будет требовать меньше процессорного времени на выборку команд и переход к блоку кода, реализующему эту команду, что положительно сказывается на производительности виртуальной машины.

Подход языково-ориентированного программирования позволяет уменьшить объем разрабатываемого кода. Один и тот же функционал может быть реализован на DSL с меньшим количеством строк, чем на языке общего назначения. При использовании DSL затраты памяти устройства будут меньше, но начиная с какого-то порогового значения объема и сложности функциональности целевой системы. На графике, изображенном на рис. 3, показано сравнение прироста объема кода при традиционном подходе и языково-ориентированном, где

ИО-1 - изначальный объем кода при традиционном подходе, ИО-2 - изначальный объем кода при языково-ориентированном подходе.

Таблица 1. Примеры виртуальных машин для ограниченных по ресурсам устройств.

Название ВМ	**Затраты Flash памяти**	**Затраты RAM**
PyMite	55 KB	8 KB
Cilix	31 KB	4 KB
Squank	5000 KB	4000 KB
uJ	80 KB	4 KB
NanoVM	8 KB	1 KB

На графике видно, что при языково-ориентированном подходе прирост объема кода с увеличением сложности системы меньше, чем при традиционном методе, но при этом требуется большего изначального объема кода. Таким образом, языково-ориентированный подход позволит уменьшить объем написанного кода и затраты памяти только с какого-то порогового значения сложности системы (П).

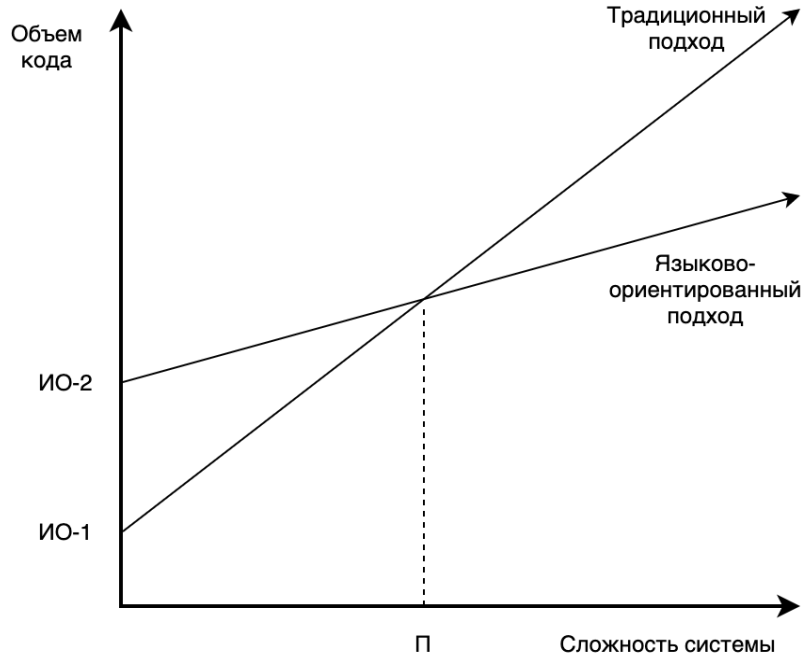

Рисунок 3. Сравнение прироста объема кода с ростом сложности системы при традиционном подходе и языково-ориентированном.

При реализации уровня виртуализации устройств и применении подхода языково-ориентированного программирования, на различных этапах жизненного цикла сети IIoT перед разработчиками будут стоять определенные задачи. На этапе проектирования необходимо определить структуру DSL и описать его грамматику, выбрать архитектуру виртуальной машины и описать ее спецификацию. На этапе реализации необходимо, соответственно, реализовать виртуальную машину, а также выбрать инструменты описания компилятора DSL в байт-код этой виртуальной машины. На этапе эксплуатации может возникать необходимость расширения функционала виртуальной машины, а также смена аппаратной платформы.

Этап проектирования

На этапе проектирования сети промышленного интернета вещей необходимо описать грамматику DSL, определить архитектуру и спецификацию виртуальной машин. Рассмотрим в качестве примера DSL, представляющий собой скриптовый язык, который позволяет определять переменные, выполнять

математические операции и выводить значения по UART. Примером такого DSL может выступать следующий код:

```
script {
    var i = 1
    var b = (i + 1) * 2

    print "I: ${i}, B: ${b}"
}
```

При реализации DSL в первую очередь необходимо определить грамматику языка. Существует несколько типов грамматик по иерархии Хомского:

- Тип 0 - Неограниченные
- Тип 1 - Контекстно-зависимые
- Тип 2 - Контекстно-независимые
- Тип 3 - Регулярные

Обычно для описания языков программирования используются контекстно-свободные и регулярные грамматики (тип 2 и 3). Контекстно-зависимые и неограниченные грамматики (тип 1 и 0) мало применяются на практике в силу своей сложности. Чтобы не усложнять процесс реализации DSL, можно использовать регулярную грамматику, поскольку процесс ее компиляции в байт-код виртуальной машины очень прост. Однако программы, написанные на регулярных языках, сложно поддерживать, поскольку они обладают слабой читаемостью. Языки, основанные на контекстно-свободной грамматике, обладают лучшей читаемостью, но процесс их компиляции сложнее. Для описания грамматики примера DSL необходимо использовать контекстно-свободную грамматику, которую можно описать при помощи расширенной нотации Бэкуса-Наура:

```
IDENTIFIER = [a-zA-Z]+.
NUMBER = 0|(-?[1-9][0-9]*).
STRING_LITERAL = "[^"]+".
OP = [+-/*].

script = "script" block.
block = "{" {expression} "}".
expression = varExpression | printExpression.
varExpression = "var" IDENTIFIER "=" valueExpression.
valueExpression = (value | valueExpressionWBR)[OP valueExpression].
valueExpressionWBR = "(" valueExpression ")".
value = NUMBER | IDENTIFIER.
printExpression = "print" STRING_LITERAL.
```

Далее необходимо определить набор команд виртуальной машины, которые содержат все необходимые функции для исполнения данного DSL. Но перед тем, как определять набор команд виртуальной машины, необходимо определить ее архитектуру. На данный момент устоялись 2 архитектуры виртуальных машин: стековые и регистровые. Основная их разница заключается в процессе исполнения команд. Исполнение команды виртуальной машины состоит из трех этапов: выборка команды, выборка операндов, выполнение вычислений.

Выборка команды включает в себя изъятие следующей инструкции из памяти ВМ и переход к соответствующему сегменту кода интерпретатора, который реализует эту инструкцию. В регистровых ВМ команды могут быть выражены меньшим количеством машинных инструкций, чем в стековых. Например, присвоение переменной a = b + c в JVM байт-коде будет представлено как ILOAD c, ILOAD b, IADD, ISTORE a

[8]. В регистровых ВМ данное выражение может быть представлено одной командой IADD a, b, c. Таким образом, количество команд в регистровых ВМ может быть значительно меньше, чем в стековых [10].

Количество обращений к памяти в регистровых машинах меньше, чем в стековых. Стековые ВМ вынуждены постоянно перекладывать значения на стек, поскольку они чаще всего моментально извлекаются следующими операциями. В регистровых ВМ значения могут храниться на стеке до момента их перезаписи, что позволяет избавиться от постоянной подгрузки значений для выполнения операций.

В то же время, трансляция языков программирования в байт-код регистровой машины требует дополнительных алгоритмов распределения регистров, поскольку их количество ограниченное. Как правило, количество переменных в коде значительно больше количества регистров, и, при компиляции, необходимо использовать алгоритмы отображения множества переменных программы на ограниченное количество регистров виртуальной машины. К тому же необходимо постоянно следить за тем, в каких регистрах находятся значения для выполнения операций. В случае стековых ВМ, операнды всегда хранятся на вершине стека.

Для примера была выбрана стековая виртуальная машина, поскольку компиляция в байт-код этой ВМ не требует дополнительных алгоритмов распределения регистров. После выбора архитектуры виртуальной машины необходимо определить набор команд, удовлетворяющий требованиям описанного DSL. Для реализации описанного выше примера DSL потребуются команды, представленные в таблице 2.

Таблица 2. Команды виртуальной машины.

Команда	Описание
ICONST	Добавление на стек константного значения
MSTORE	Сохранение значения из стека в памяти
MLOAD	Выгрузка значения из памяти в стек
IADD	Суммирование двух значений на стеке
ISUB	Вычитание двух значений на стеке
IMULT	Произведение двух значений на стеке
IDIV	Целочисленное деление двух значений на стеке
PRNTCHR	Вывод символа по USART
PRNTVAR	Вывод значения из памяти по USART
HALT	Завершение работы программы

Этап реализации

После определения спецификации виртуальной машины и грамматики DSL необходимо реализовать компилятор DSL в байт-код виртуальной машины. Компиляция языков программирования состоит из следующих обязательных этапов: лексический анализ, синтаксический анализ, генерация кода. Для облегчения разработки компилятора DSL можно использовать генератор лексических и синтаксических анализаторов. Существует множество генераторов лексических и синтаксических анализаторов, основными из них являются: lex, yacc, antlr4. Также существуют GNU версии lex и yacc: flex и bison. От выбора генераторов зависит сложность разработки компилятора DSL. Основные отличия antlr4 от lex и yacc заключаются в формате описания грамматик. Antlr4 поддерживает грамматики в расширенной нотации Бэкуса-Наура, в то время как yacc поддерживает только стандартную нотацию, что усложняет описание грамматик. Также, при использовании lex, правила лексического анализа задаются при помощи регулярных выражений, что достаточно в большинстве случаев, но усложняет процесс описания правил. Antlr4 позволяет описывать правила лексического анализа при помощи контекстно-свободных выражений, что упрощает определение некоторых общих элементов [12]. Также существуют системы метапрограммирования, например Jetbrains MPS. Эта система реализует парадигму языково-ориентированного программирования, но основной ее недостаток в том, что компилятор DSL, генерируемый этой системой, нельзя использовать вне ее. В этом случае разработчики будут вынуждены использовать Jetbrains MPS для

программирования устройств IIoT. Для реализации компилятора примера DSL был выбран antlr4 из-за простоты описания грамматик и возможности переиспользования компилятора в других системах или приложениях. Пример грамматики DSL в нотации antlr4:

```
WS : [\n\t\r ]+ -> skip;
IDENTIFIER : [a-zA-Z]+;
NUMBER : '0'|'-'?[1-9][0-9]*;
STRING_LITERAL : '"' ~["] '"'
OP : '+' | '-' | '*'  | '/';

script : 'script' block;
block : '{' expression* '}';
expression : varExpression | printExpression;
varExpression : 'var' IDENTIFIER '=' valueExpression;
valueExpression : (value | valueExpressionWBR) (OP valueExpression)*;
valueExpressionWBR : '(' valueExpression ')';
value : NUMBER | IDENTIFIER;
printExpression = 'print' STRING_LITERAL;
```

По данной грамматике antlr4 генерирует лексический и синтаксический анализатор, который можно использовать в дальнейшем для компиляции. Исходный код DSL подается на вход лексическому анализатору, который разбивает его на набор токенов. Далее этот набор токенов подается на вход синтаксическому анализатору для проверки корректности синтаксиса программы и формирования абстрактного синтаксического дерева. Абстрактное синтаксическое дерево через генератор кода при помощи алгоритма сортировочной станции преобразуется из инфиксной нотации в постфиксную и формируется байт-код виртуальной машины. На рис. 4 представлен процесс компиляции исходного кода DSL в байт-код виртуальной машины.

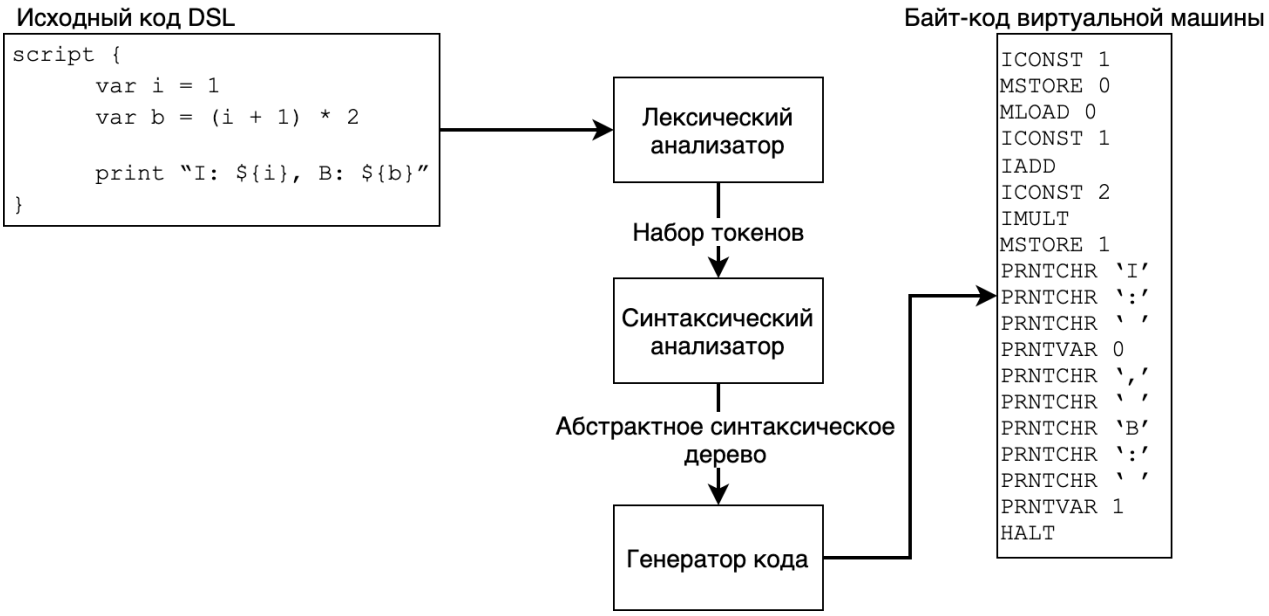

Рисунок 4. Процесс компиляции исходного кода DSL в байт-код виртуальной машины.

Далее байт-код загружается в память виртуальной машины для исполнения. Для реализации самой простой стековой ВМ потребуется: стек, указатель на инструкцию (ip), выделенная память для хранения байт-кода и массив для хранения значения переменных (memory). Также потребуются вспомогательные методы, описанные в таблице 3.

Таблица 3. Вспомогательные методы виртуальной машины.

Сигнатура метода	Описание
void push(int value)	Добавление значения на вершину стека
int pop(void)	Изъятие значения с вершины стека
void fetch(void)	Изъятие следующей команды байт-кода
void sendSymbol(char symbol)	Оправка символа по UART
void sendMessage(char[] message)	Отправка массива символов по UART
char[] toString(int value)	Конвертация целочисленного значения в массив символов

Основной цикл выполнения команды виртуальной машины состоит из следующих этапов: выборка команды, декодирование команды и переход к области кода, реализующему эту команду. Пример реализации виртуальной машины:

```
while (ip < BYTECODE_SIZE) {
      switch (fetch()) {
            case ICONST : iconst(); break;
            case MSTORE : mstore(); break;
            case MLOAD: mload(); break;
            case IADD : iadd(); break;
            case isub : isub(); break;
            case IMULT : imult(); break;
            case IDIV : idiv(); break;
            case PRNTCHR : prntchr(); break;
            case PRNTVAR : prntvar(); break;
            case HALT : halt(); break;
      }
}

void iconst()  { push(fetch()); }
void mstore()  { memory[fetch()] = pop(); }
void mload()   { push(memory[fetch()]); }
void iadd()    { push(pop() + pop()); }
void isub()    { int op2 = pop(); int op1 = pop(); push(op1 - op2); }
void imult()   { push(pop() * pop()); }
void idiv()    { int op2 = pop(); int op1 = pop(); push(op2 / op1); }
void prntchr() { sendSymbol((char) pop()); }
void prntvar() { sendMessage(toString(memory[fetch()])); }
void halt()    { ip = BYTECODE_SIZE; }
```

Для оптимизации виртуальной машины можно использовать такие техники как: шитый код, супер-инструкции и уменьшение размера байт-кода. Шитый код позволяет уменьшить количество машинных инструкций, необходимых для перехода к следующей команде байт-кода. Выполнение команд при помощи оператора switch в среднем требует 9-10 машинных инструкций, в то время как шитый код позволяет уменьшить их количество до 3-4. Супер-инструкции позволяют заменять часто используемые блоки байт-кода в отдельные команды, что уменьшает количество переходов между первым и последним элементом заменяемого блока [11]. Также на уровне компиляции DSL в байт-код можно добавить дополнительный этап оптимизации, который позволит уменьшить общий размер исполняемого кода. Например, идущие друг за другом команды MSTORE N

и MLOAD N, можно объединить в одну команду MSTORE_P N. Данная команда сохранит значение из стека в памяти и оставит это значение на вершине стека.

Этап эксплуатации

На этапе эксплуатации часто возникает необходимость расширения существующего функционала виртуальной машины и, в крайних случаях, смена аппаратной платформы.

Для определения новых функций виртуальной машины можно дополнить ее новыми командами как низкого, так и высокого уровня абстракции. Но, в случае, когда есть необходимость запускать уже скомпилированный код DSL на новых версиях виртуальной машины, важно поддерживать обратную совместимость, поскольку логика работы уже скомпилированного кода не должна меняться в зависимости от версии виртуальной машины. Зачастую определение новых команд происходит путем модификации исходного кода виртуальной машины. Но, если нет возможности физического доступа к устройству для замены прошивки, можно в виртуальной машине реализовать интерпретатор команд, который позволит объявлять новые команды виртуальной машины удаленно. Но такой подход требует дополнительные затрат на прием и интерпретацию команд.

Необходимость смены аппаратной платформы чаще всего возникает при вертикальном масштабировании. Вертикальное масштабирование может осуществляться как путем добавления ресурсов устройства, так и сменой аппаратной платформы (например, замена Arduino на STM32 даст ощутительный прирост производительности). В таком случае разработчики вынуждены адаптировать код виртуальной машины под новую платформу. И, если при этом удается полностью сохранить спецификацию виртуальной машины, то уже скомпилированный код DSL будет функционировать на новой платформе так же, как и раньше.

Вывод

Архитектура любой сети промышленного интернета вещей включает в себя уровень устройства, обеспечивающий функциональность устройств IIoT. При программировании данных устройств необходимо учитывать специфики их аппаратной платформы, что не позволяет переиспользовать уже существующее программное обеспечение на других платформах. Также проблемой разработки уровня устройства сети IIoT является ограниченность ресурсов устройств. При росте сложности системы увеличивается объем кода и затраты памяти устройств, что может привести к нехватке памяти для хранения программного обеспечения.

Для решения проблемы платформозависимости ПО можно добавить дополнительный уровень в архитектуру сети - уровень виртуализации устройств. Этот уровень должен предоставлять не зависящие от аппаратной платформы виртуальные функции устройств, что позволит писать программное обеспечение под виртуальную платформу, аппаратная часть которой может быть заменена, не затрагивая непосредственно ПО. Одним из способов реализации уровня виртуализации устройств могут выступать виртуальные машины, а конкретно среды языков программирования.

Виртуальные машины позволяют абстрагироваться от специфик аппаратной платформы устройства. Существует множество реализаций виртуальных машин для устройств с ограниченными ресурсами, но они, зачастую, программируются при помощи языков общего назначения и не подходят для применения подхода языково-ориентированного программирования. Реализация виртуальной машины под конкретную предметную область может облегчить применение подхода языково-ориентированного программирования, поскольку такие машины могут предоставлять команды как низкого уровня абстракции, так и высокого. Подход языково-ориентированного программирования позволяет уменьшить прирост объема кода с ростом сложности системы.

При внедрении предметно-ориентированных виртуальных машин в жизненный цикл промышленного интернета вещей на различных этапах перед разработчиками будут стоять определенные задачи. На этапе проектирования сети необходимо описать грамматику DSL, выбрать архитектуру виртуальной машины и описать ее спецификацию. Существует 4 типа грамматик по иерархии Хомского, но для описания DSL практически всегда используется контекстно-свободные или регулярные грамматики. Процесс компиляции программ, написанных на регулярных языках, осуществляется проще, чем на контекстно-свободных, однако они обладают слабой читаемостью, что усложняет поддержку ПО. Также выбор архитектуры виртуальной машины может существенно сказаться на ее производительности. На данный момент считаются устоявшимися 2 архитектуры: стековая и регистровая. В регистровых ВМ команды могут выполнять больше функций, чем в стековых, что уменьшает размер байт-кода, к тому же в регистровых ВМ меньше обращений к памяти, поскольку значения в регистрах хранятся до момента их перезаписи. Но процесс компиляции исходного кода DSL в байт-код стековой машины намного проще, поскольку не требует дополнительных алгоритмов распределения регистров, так как их количество ограниченное.

На этапе реализации сети необходимо, непосредственно, реализовать виртуальную машину и компилятор DSL в байт-код этой машины. Для облегчения реализации компилятора DSL можно воспользоваться генераторами лексических и синтаксических анализаторов. Например, lex (GNU альтернатива - flex), yacc (GNU

альтернатива - bison), antlr4. Их основное отличие в формате описания грамматик. Yacc поддерживает грамматики только в стандартной нотации Бэкуса-Наура, что усложняет ее описание, в то время как antlr4 позволяет описывать грамматики в расширенной нотации. Для описания правил лексического анализа lex использует регулярные выражения, чего достаточно в большинстве случаев, но усложняет процесс описания правил. Antlr4 использует контекстно-свободные выражения для описания правил лексического анализа. При реализации виртуальной машины можно использовать различные техники оптимизации: шитый код, супер-инструкции, уменьшение размера байт-кода.

На этапе эксплуатации может возникать необходимость расширения функционала виртуальной машины, и, в крайних случаях, смена аппаратной платформы. Расширить функции виртуальной машины можно, непосредственно, изменив исходный код ВМ, либо реализовать интерпретатор команд, который позволит удаленно описывать новые функции без непосредственного физического контакта с устройством, но такой подход требует дополнительных затрат ресурсов на прием и интерпретацию команд. При необходимости смены аппаратной платформы разработчики будут вынуждены адаптировать большую часть кода виртуальной машины под новую платформу.

Помимо преимуществ, виртуальные машины также имеют ряд недостатков. Для работы виртуальных машин требуются дополнительные ресурсы процессора на выборку команд и переход к блоку кода, реализующему эту команду, что в целом замедляет работу программ. В этом случае можно применять различные техники оптимизации, описанные выше. Также при применении языково-ориентированного подхода, требуются дополнительные затраты на реализацию компилятора DSL. Для облегчения разработки компилятора DSL можно воспользоваться генераторами лексических и синтаксических анализаторов. Также недостатком виртуальных машин является необходимость поддержки обратной совместимости. Новые версии виртуальной машины не должны нарушать логику работы уже скомпилированных программ.

Таким образом следует, что применение виртуальных машин чаще всего уместно для систем IIoT с большой кодовой базой, в процессе эксплуатации которых планируется обновление программного обеспечения устройств или смена аппаратной платформы.

Список литературы

1. Москаленко Т. А., Киричек Р. В., Бородин А. С. Архитектуры промышленного Интернета Вещей // Информационные технологии и телекоммуникации — 2017 — Том 5. № 4. — С. 49–56.
2. Auler R., Millani C.E., Brisighello A., Linhares A., Borin E. Handling IoT platform heterogeneity with COISA, a compact OpenISA virtual platform // Concurrency and Computation. Practice and Experience — 2017 — Vol.29, Issue 22
3. Boyes H., Hallaq B., Cunningham J., Watson T. The industrial internet of things (IIoT): An analysis framework // Computers in industry — 2018 — Vol. 101 — P. 1-12
4. Davis R., Sanden B.I., Laplante P.A. A real-time virtual machine implementation for small microcontrollers // Innovations in Systems and Software Engineering — 2012 — Vol. 8 — P. 223-241
5. Deursen A., Klint P., Visser J. Domain-Specific Languages: An Annotated Bibliography // SIGPLAN Notices —2000 — Vol. 35, Issue 6 — P. 26-36.
6. Industrial Internet of Things Consortium Reference Architecture [Электронный ресурс] // Industrial Internet Consortium — URL: https://www.iiconsortium.org/IIC_PUB_G1_V1.80_2017-01-31.pdf (дата обращения: 12.04.2021)
7. ISO/IEC 30141:2018 Internet of Things (IoT) — Reference Architecture// [Электронный ресурс] International Organization for Standardization — URL: https://www.iso.org/standard/65695.html (дата обращения: 04.04.2021)
8. Lindholm T., Yellin F., Bracha G., Buckley A. The Java Virtual Machine Specification [Электронный ресурс] // Oracle documentation — URL: https://docs.oracle.com/javase/specs/jvms/se8/html/index.html (дата обращения: 10.04.2021)
9. Martin S. Design and Implementation of an Efficient Stack Machine. 19th IEEE International Parallel and Distributed Processing Symposium. Денвер, Колорадо. 2005. 472с.
10. Shi Y., Casey K., Ertl M. A., Gregg D. Virtual machine showdown: Stack versus registers // ACM Transactions on Architecture and Code Optimization — 2008 — Vol.4, Issue 4
11. Stefan B. Virtual-Machine Abstraction and Optimization Techniques // Electronic Notes in Theoretical Computer Science – 2009 Vol. 253, Issue 5 – P. 3-14
12. Tomassetti G. Why you should not use (f)lex, yacc and bison // [Электронный ресурс] Strumenta — URL: https://tomassetti.me/why-you-should-not-use-flex-yacc-and-bison/ (дата обращения: 15.04.2021)

References

1. Moskalenko T. A., Kirichek R. V., Borodin A. S. Architectures of Industrial Internet of things // Information technology and telecommunications — 2017 — Vol. 5., № 4. — P. 49–56.
2. Auler R., Millani C.E., Brisighello A., Linhares A., Borin E. Handling IoT platform heterogeneity with COISA, a compact OpenISA virtual platform // Concurrency and Computation. Practice and Experience — 2017 — Vol.29, Issue 22
3. Boyes H., Hallaq B., Cunningham J., Watson T. The industrial internet of things (IIoT): An analysis framework // Computers in industry — 2018 — Vol. 101 — P. 1-12
4. Davis R., Sanden B.I., Laplante P.A. A real-time virtual machine implementation for small microcontrollers // Innovations in Systems and Software Engineering — 2012 — Vol. 8 — P. 223-241
5. Deursen A., Klint P., Visser J. Domain-Specific Languages: An Annotated Bibliography // SIGPLAN Notices —2000 — Vol. 35, Issue 6 — P. 26-36.
6. Industrial Internet of Things Consortium Reference Architecture [Electronic resource] // Industrial Internet Consortium — URL: https://www.iiconsortium.org/IIC_PUB_G1_V1.80_2017-01-31.pdf (access date: 12.04.2021)
7. ISO/IEC 30141:2018 Internet of Things (IoT) — Reference Architecture// [Electronic resource] International Organization for Standardization — URL: https://www.iso.org/standard/65695.html (access date: 04.04.2021)
8. Lindholm T., Yellin F., Bracha G., Buckley A. The Java Virtual Machine Specification [Electronic resource] // Oracle documentation — URL: https://docs.oracle.com/javase/specs/jvms/se8/html/index.html (access date: 10.04.2021)
9. Martin S. Design and Implementation of an Efficient Stack Machine. 19th IEEE International Parallel and Distributed Processing Symposium. Denver, CO. 2005. 472p.
10. Shi Y., Casey K., Ertl M. A., Gregg D. Virtual machine showdown: Stack versus registers // ACM Transactions on Architecture and Code Optimization — 2008 — Vol.4, Issue 4
11. Stefan B. Virtual-Machine Abstraction and Optimization Techniques // Electronic Notes in Theoretical Computer Science – 2009 Vol. 253, Issue 5 – P. 3-14
12. Tomassetti G. Why you should not use (f)lex, yacc and bison // [Electronic resource] Strumenta — URL: https://tomassetti.me/why-you-should-not-use-flex-yacc-and-bison/ (access date: 15.04.2021)