

**ADAM: децентрализованная параллельная компьютерная  
архитектура с**

**Быстрая миграция потоков и данных и единая  
аппаратная абстракция<sub>к</sub>**

Эндрю «Банни» Хуанг

Подано на кафедру электротехники и информатики в порядке  
частичного выполнения требований к ученой степени доктора  
философии в

МАССАЧУСЕТСКИЙ ТЕХНОЛОГИЧЕСКИЙ ИНСТИТУТ

Июнь 2002 г.

с Массачусетский технологический институт 2002. Все права защищены.

Автор . ...

Кафедра электротехники и информатики  
24 мая 2002 г.

Сертифицировано. ...

Томас Ф. Найт-младший.  
Старший научный сотрудник  
Руководитель диссертации

Принято . ...

Артур С. Смит Председатель Комитета по делам  
аспирантов факультета

**ADAM: децентрализованная параллельная компьютерная архитектура с  
быстрой миграцией потоков и данных, а также единой аппаратной  
абстракцией**

к

Эндрю «Банни» Хуанг

Представлено на кафедру электротехники и информатики 24 мая 2002  
г. в порядке частичного выполнения требований к ученой степени  
доктора философии

**Абстрактный**

Бешеный темп закона Мура приводит компьютерную архитектуру в область, где скорость света является доминирующим фактором в задержках системы. Количество тактовых циклов, охватывающих чип, увеличивается, в то время как количество битов, к которым можно получить доступ в течение тактового цикла, уменьшается. Следовательно, становится все труднее скрывать задержку. Одним из альтернативных решений является уменьшение задержки путем миграции потоков и данных, но накладные расходы существующих реализаций ранее делали миграцию неработоспособным решением до сих пор.

Я представляю архитектуру, реализацию и механизмы, которые снижают накладные расходы на миграцию до такой степени, что миграция становится жизнеспособным дополнением к другим механизмам сокрытия задержек, таким как многопоточность. Архитектура является абстрактной и предоставляет программистам простую, унифицированную мелкозернистую многопоточную модель параллельного программирования с неявным управлением памятью. Другими словами, пространственная природа и детали реализации (например, количество процессоров) параллельной машины полностью скрыты от программиста. Авторы компиляторов поощряются к разработке языков программирования для машины, которые помогают программисту выражать свои идеи в терминах объектов, поскольку объекты демонстрируют присущую им физическую локальность данных и кода. Реализация машины затем может использовать эту локальность для автоматического распределения данных и потоков по физической машине с помощью набора высокопроизводительных механизмов миграции.

Реализация этой архитектуры может перенести нулевой поток за 66 циклов — улучшение более чем в 1000 раз по сравнению с предыдущей работой. Производительность также хорошо масштабируется; время, необходимое для перемещения типичного потока, всего в 4–5 раз больше, чем у нулевого потока. Производительность миграции данных аналогична и линейно масштабируется с размером блока данных. Поскольку производительность механизма миграции находится на одном уровне с производительностью кэша L2, реализация, смоделированная в моей работе, не имеет кэшей данных и вместо этого полагается на многопоточность и механизм миграции для сокрытия и сокращения задержек доступа.

Научный руководитель: Томас Ф. Найт, младший.  
Должность: Старший научный сотрудник

## Благодарности

Я хотел бы поблагодарить своих родителей за всю их любовь и поддержку на протяжении многих лет, а также за их неизменную поддержку и веру в мою способность закончить обучение по программе получения степени.

Я также хотел бы поблагодарить мою замечательную, любящую, заботливую девушку Никки Джастис за всю ее поддержку, мотивацию, терпение, редактирование, пайку, обсуждение и доработку идей, готовку, уборку, стирку, отвозку меня в кампус, массаж запястий, знание, когда мне нужно заняться инструментами, а когда мне нужно сделать перерыв, терпимость к моему графику сна в 7 утра и за то, что она позволила мне устроить беспорядок в ее комнате и занять ее диван всем моим компьютерным оборудованием.

Я хотел бы поблагодарить всех моих друзей за их поддержку на протяжении многих лет и за то, что сделали последнее десятилетие в MIT – и мой первый шаг в реальный мир – захватывающим, веселым и полезным опытом. Пусть начнется спешка... и пусть она никогда не закончится.

Эта диссертация никогда бы не появилась, если бы не Aries Research Group (в порядке старшинства): Том Найт, Норм Марголус, Джереми Браун, Дж. П. Гроссман, Джози Аммер, Майк Филлипс, Пегги Чен, Бобби Вудс-Корвин, Бен Вандивер, Том Клири, Доминик Риццо и Брайан Гинзбург. Том Найт, в частности, был для меня образцом для подражания с тех пор, как я пришел в лабораторию; он является бесконечным источником вдохновения и знаний, и давал бесценные указания, советы и поддержку. Он блестящий и дальновидный, но в то же время скромный и очень доступный, и всегда готов ответить на мои вопросы, какими бы глупыми или тупыми они ни были. Мне также очень нравится его политика невмешательства в отношении управления группой; я действительно ценю интеллектуальную свободу, которую Том принес в группу, и его огромную веру во все наши способности управлять и организовывать себя, а также «идти вперед и думать великие мысли». Джереми Браун и Дж. П. Гроссман также были бесценны за их хорошие идеи, живое общение и доработку идей. Джереми изобрел идемпотентный сетевой протокол, используемый в этой диссертации, и его превосходная диссертационная работа по новым методам параллельного программирования и масштабируемой параллельной сборке мусора заполняет многие важные пробелы в моей диссертации. Дж. П. и Джереми также

разработали представление возможностей с помощью SQUIDS, которое является центральным в моей диссертации. Я также полагался на превосходную работу Дж. П. по исследованию и характеристике различных сетевых топологий и схем; я использовал многие из его результатов в своей реализации. Бобби Вудс-Корвин, Пегги Чен, Брайан Гинзбург и Доминик Риццо были бесценны в разработке реализации сети. Без них мне было бы нечего показать для этой диссертации, кроме кучи кода Java. Два поколения диссертаций M.Eng и два UROP — это большая работа! Норм Марголус также помог заложить основы архитектуры, работая над пространственными клеточными автоматами и встроенными процессорами DRAM.

Наконец, Андре ДеХон, хотя официально и не входил в группу, во многом сыграл очень важную роль в моей работе. Эта работа во многом опирается на его более раннюю работу в MIT по сети METRO. Андре также давал бесценные советы и отзывы во время своих визитов из Калтеха.

Я также хотел бы выразить особую благодарность Бену Вандиверу. С момента создания ADAM и Q-Machine Бен предоставил бесценные идеи и отзывы. Диссертация была бы неполной без его синергии как автора компиляторов, программиста и высококлассного хакера программного обеспечения. Я также благодарю его за его энтузиазм и веру в архитектуру; его позитивная энергия была необходима, чтобы не дать мне отчаяться. Он не только помог разработать модель программирования для машины, но и написал два компилятора для машины по ходу дела. Он также сыграл важную роль в кодировании и отладке бенчмарков, используемых в разделе результатов моей диссертации.

Крсте Асанович и Ларри Рудольф также оказали большое влияние на эту диссертацию. Крсте — источник знаний и необычайно острого понимания даже самых тонких архитектурных деталей. Ларри открыл мне глаза на область конкурентного анализа и онлайн-алгоритмов, чего я бы никогда не стал делать иначе. Я также ценю критический обзор, предоставленный как Крсте, так и Ларри.

Я также хотел бы поблагодарить моих друзей из Mobilian Corporation, в частности Роба Гилмора, МэриДжо Неттлз, Тодда Саттона и Роба Уэнтворта, за их понимание, терпение и поддержку в получении мной степени.

Я благодарю корпорацию Xilinx за щедрый дар в виде множества высокопроизводительных ПЛИС и инструментов проектирования для проекта, которые использовались для реализации прототипа сети и узлов процессора. Я также хотел бы поблагодарить корпорацию Intel и Silicon Spice за предоставление стипендий и оборудования, которые позволили мне закончить мою работу. Корпорация Sun и Borland также предоставили мне Java и JBuilder бесплатно, но ценность этих инструментов нельзя недооценивать. Эта работа также финансировалась Исследовательской лабораторией BBC, номер соглашения F30602-98-1-0172, «Технология активных баз данных».

Я мог бы продолжать, но, к сожалению, не хватит места, чтобы назвать всех, кто помог мне с моей диссертацией. Это всем, кто внес неоценимый вклад и руководство в мою

диссертацию – спасибо всем. Я в долгу перед миром, и я могу только надеяться, что когда-нибудь внесу достойный вклад.

Наконец, все ошибки в этой диссертации — мои.

# Содержание

<b>1 Введение .....</b>	<b>14</b>
1.1 Вклады .....	15
1.2 Организация этой работы .....	16
<b>2 Предыстория .....</b>	<b>18</b>
2.1 Методы управления задержками .....	18
2.1.1 Сокращение задержки .....	18
2.1.2 Скрытие задержки .....	21
2.2 Механизмы миграции .....	23
2.2.1 Обсуждение .....	25
2.3 Архитектурная родословная .....	26
2.3.1 Поток данных .....	26
2.3.2 Разделенный доступ/выполнение .....	29
2.3.3 Процессор в памяти (PIM) и чип-мультипроцессоры (CMP) .....	31
2.3.4 Архитектуры кэш-памяти .....	32
<b>3 Овна Децентрализованная Абстрактная Машина .....</b>	<b>33</b>
3.1 Введение в ADAM на примере кода .....	33
3.1.1 Основы .....	34
3.1.2 Соглашение о вызове .....	34
3.1.3 Распределение памяти и доступ к ней .....	36
3.2 Модель программирования .....	38
3.2.1 Темы .....	39
3.2.2 Очереди и сопоставления очередей .....	40
3.2.3 Модель памяти .....	41
3.2.4 Взаимодействие с памятью .....	43

<b>4 Механизм миграции в децентрализованной вычислительной среде .....</b>	<b>45</b>
4.1 Введение.....	45
4.2 Предыстория.....	46
4.2.1 Архитектуры, которые напрямую решают проблему миграции .....	46
4.2.2 Механизмы мягкой миграции .....	48
4.2.3 Среды программирования и алгоритмы онлайн-миграции.....	52
4.3 Реализация механизма миграции.....	55
4.3.1 Механизм удаленного доступа к памяти .....	56
4.3.2 Механизм миграции.....	59
4.3.3 Миграция данных.....	60
4.3.4 Миграция потоков.....	63
4.4 Проблемы и наблюдения механизма миграции .....	69
4.4.1 Общие замечания .....	69
4.4.2 Проблемы с производительностью .....	71
<b>5 Реализация ADAM: Аппаратное обеспечение и моделирование .....</b>	<b>74</b>
5.1 Введение.....	74
5.2 Организация высокого уровня .....	75
5.3 Конечный узел.....	76
5.3.1 Процессорный узел .....	77
5.3.2 Узел памяти .....	83
5.4 Физическая конструкция.....	85
5.4.1 Технологические предположения.....	85
5.4.2 Описание дизайна .....	88
<b>6 Характеристика машин и миграции .....</b>	<b>92</b>
6.1 Основные результаты производительности Q-Machine .....	92
6.1.1 Производительность памяти .....	94
6.1.2 Базовая производительность сетевых операций .....	94
6.2 Эффективность миграции и контроль миграции: простые случаи.....	96
6.2.1 Тест производительности двух потоков .....	96

6.2.2 Тест производительности потоков и памяти .....	102
6.3 Случаи применения.....	105
6.3.1 Применение быстрой сортировки на месте .....	106
6.3.2 Тест умножения матриц .....	110
6.3.3 Тест N-тела.....	114
<b>7 Выводов и Дальнейшая Работа .....</b>	<b>119</b>
7.1 Выводы.....	119
7.2 Будущая работа .....	120
7.2.1 Улучшенные алгоритмы управления миграцией .....	121
7.2.2 Языки и компиляторы.....	122
7.2.3 Аппаратная реализация .....	123
7.2.4 Транзакции.....	123
7.3 Заключительные замечания.....	124
<b>Акронимы .....</b>	<b>125</b>
<b>Подробности Б АДАМ .....</b>	<b>131</b>
Б.1 Типы данных .....	132
В.2 Форматы инструкций .....	138
В.3 Формат возможностей.....	141
В.4 Uber-возможности и многозадачность” .....	144
В.5 Обработка исключений .....	145
<b>С Q-Подробности о машине .....</b>	<b>148</b>
С.1 Подробности реализации файла очереди .....	148
С.1.1 Физическая конструкция .....	149
С.1.2 Конечный автомат .....	153
С.2 Сетевой интерфейс .....	156
С.3 Топология сети и реализация .....	162
<b>Коды операций D.....</b>	<b>166</b>
D.1 Общие примечания.....	166
D.2 Ленивые инструкции.....	166



D.3 Краткое изложение инструкции.....	167
---------------------------------------	-----

# Список фигур

1-1 Обзор уровней абстракции в этой диссертации. Couatl и People — компиляторы автор Бен Вандивер. ....	20
2-1 Доступная область кристалла в верхнем слое металла, где площадь измеряется в шести транзисторах Ячейки SRAM. Непосредственно из [АНКВ00] .....	24
2-2 Иллюстрация проблемы ложного обмена. ....	28
3-1 Демонстрация копирования/затирания (@) модификатор. . . . .	36
3-2 Простой пример кода, демонстрирующий связывание процедур, порождение потоков, память выделение памяти и доступ к ней. ...	37
3-3 Состояния потока после создания потока и связывания процедур. ....	39
3-4 Состояния потока после выделения памяти и доступа к ней. ...	40
3-5 Программная модель ADAM .....	41
3-6 Структура потока ADAM .....	42
3-7 Высокоуровневая разбивка формата возможностей ADAM. Подробная разбивка на уровне битовак- С результатами каждого поля можно ознакомиться в приложении В. ....	44
4-1 Формат теневого пространства возможностей удаленной памяти в локальном виртуальном пространстве памяти. 57	
4-2 Представление системного уровня разрешения удаленных запросов памяти. . .58 .....	
4-3 Подробности обработки удаленных и локальных данных Запросы на биржу. . .58 .....	
4-4 Механизм временной заморозки запросов памяти. ....	61
4-5 Обработка мигрировавшего запроса EXCH с временными двунаправленными указателями. . . .	63
4-6 Протокол линии передачи для обработки обновлений указателя пересылки на сопоставленных потоках коммуникации. ....	68

4-7	Обзор схемы распространения данных по запросу. ....	70
5-1	Части реализации Q-машины. Теги идентификаторов узлов одинаковы по всей машине, поэтому сетевое пользовательское оборудование адресуемо, как любой процессор или узел памяти. ....	73
5-2	Высокоуровневая структурная схема листового узла. ....	74
5-3	Деталь процессорного узла. ....	75
5-4	Гибридная структура списка планировщика/L-кэша. На этой диаграмме c42 и c10 готовы к запуску и пересылке в рабочую очередь; поскольку значения для c55:q12 и c4:q4 поступают через NI, они будут повышены до работоспособного статуса. ...	78
5-5	Высокоуровневая структурная схема узла памяти. ....	79
5-6	Упаковка и интеграция для двухслойного кремниевого высокопроизводительного чипа мультипроцессионарий. ....	82
5-7	Рисунок макета сетевого уровня. ....	83
5-8	Гипотетическая компоновка одного процессорного узла. ....	84
5-9	Гипотетическая компоновка чипа процессора плитки. ....	85
6-1	Скриншот ASS, выполняющего 64-узловой векторный обратный регрессионный тест. Слева обзор машины; справа — окно отладчика потоков. ....	88
6-2	Синтетический бенчмарк двух потоков. Связь происходит по дугам; зависимость от данных принудительно устанавливается путем печати входящих данных. ....	91
6-3	Код, используемый для двухпоточного теста. ....	92
6-4	Измеренное ускорение в зависимости от расстояния миграции для теста Two Threads. ...	93
6-5	Форма кривой—. ....	95
6-6	Длина последовательности сообщений, необходимая для амортизации различных накладных расходов на миграцию ( ). Базовый уровень — два сообщения на итерацию для двухпоточного теста — также отмечено на графике. ...	96
6-7	Синтетический бенчмарк потока и памяти. Связь происходит по дуги; зависимость данных принудительно устанавливается путем печати входящих данных. ....	97
6-8	Код, используемый для теста производительности потоковой памяти. ....	98

6-9	Ускорение миграции в зависимости от времени принятия решения о миграции и объема памяти	
	тест потоков и памяти. ...	99
6-10	циклов на итерацию для теста Thread-Memory. в обоих случаях. ....	99
6-11	Метод объекта для теста быстрой сортировки, написанный на языке People. . .	
.....		101
6-12	Распределение времени миграции, используемое в тесте быстрой сортировки. .	102
.....		
6-13	График метрики нагрузки по сравнению со временем для теста быстрой сортировки с использованием и без использования	
	Балансировка нагрузки . ...	103
6-14	График теста быстрой сортировки с балансировкой нагрузки с наложенными событиями миграции. . .	105
6-15	Часть теста умножения потоковой матрицы, написанная на People. ....	106
6-16	График времени, необходимого для итерации умножения матрицы 100x100 на различные условия миграции и стили кодирования. ....	108
6-17	График времени, необходимого для итерации умножения матрицы 15x15 на различные микросхемы. Условия градации и стили кодирования. ....	108
6-18	График первых нескольких временных шагов выходного сигнала теста N-Body. ....	109
6-19	Внутренний цикл кода эталонного теста N-Body. ....	111
6-20	График времени, необходимого для каждого временного шага моделирования 12-тел N-тел, запущенного на 64-узловая Q-машина. ....	111
Б-1	Форматы данных, поддерживаемые ADAM . ....	124
Б-2	Подробности полей тегов и флагов . ...	125
Б-3	Формат кодов операций ADAM . ....	127
Б-4	Формат возможностей ADAM . ....	129
Б-5	Обзор обработки исключений . ....	133
С-1	Реализация VQF с 3 портами записи и 3 портами чтения. $pq \neq \#$ физические регистры . Q-	
	Подробности кэша опущены для ясности. ....	136

C-2	Элементарная ячейка PQF. ...	137
C-3	Блок-схема ответа на запрос чтения PQF .....	140
C-4	Блок-схема ответа на запрос записи PQF .....	142
C-5	Подробная информация о сетевом интерфейсе.. .....	144
C-6	Подробное описание протокола идемпотентности и надежной доставки данных для одной транзакции. Линии серого цвета — это линии «повторных попыток», которые не возникли бы в идеальных условиях. ....	146
C-7	Подробности форматов пакетов. Обратите внимание, что в заголовках cID назначения/источника и очереди очень важно, чтобы идентификатор процессора находился в MSB и был совмещен с полем адреса, поскольку реализации могут вставлять биты между полями адреса и PID для увеличения количества маршрутизируемых узлов процессора или для увеличения количества памяти на узел. ....	147
	Формат D-1 qb для инструкции PARCEL .....	240

# Список таблиц

5.1 Экстраполированные технологические параметры на 2010 год. Все значения из [CI00a], если не указано иное.

в противном случае отмечено. ....	81
A.1 Таблица сокращений .....	120
A.2 Таблица сокращений, продолжение .....	121

# Глава 1

## Введение

Невозможно подделать пропускную способность памяти, которой нет.

—*Сеймур Крэй о том, почему у Cray-1 не было тайников*

Большинство механизмов миграции данных и потоков на сегодняшний день медленны по сравнению с другими методами управления задержкой. В этой диссертации представлена архитектура ADAM, которая обеспечивает простую аппаратную реализацию миграции данных и потоков. Эта реализация снижает накладные расходы на миграцию до точки, где она сопоставима с другими аппаратными методами управления задержкой, такими как кэширование.

Миграция данных полезна для сокращения задержек доступа в ситуациях, когда рабочий набор больше кэша. Она также полезна для сокращения или перераспределения сетевого трафика в ситуациях, когда горячие точки вызваны конкуренцией за несколько объектов данных. Миграция данных также может использоваться для эмуляции функции кэшей в системах, в которых нет кэшей данных.

Миграция потоков полезна для сокращения задержек доступа в ситуациях, когда несколько потоков борются за один фрагмент данных. Как и миграция данных, она также полезна в ситуациях, когда горячие точки можно устранить путем перераспределения источников и пунктов назначения сетевого трафика. Миграция потоков также полезна для балансировки нагрузки, особенно в ситуациях, когда конкуренция за память низкая.

Миграция данных и потоков может использоваться вместе для управления задержками доступа в ситуациях, когда множество потоков обмениваются информацией непредсказуемым образом среди множества фрагментов данных, как это может быть в случае с корпоративной базой данных. Миграция данных и потоков также может использоваться для повышения надежности системы, если сбои можно предсказать достаточно заранее, чтобы сбойный узел мог быть очищен от своего содержимого.

## 1.1 Вклады

Основной вклад моей диссертации — быстрый, малозатратный механизм миграции данных и потоков. С точки зрения циклов процессора механизм, описанный в моей диссертации, представляет собой более чем 1000-кратное увеличение производительности по сравнению с предыдущими программными механизмами миграции. В результате накладные расходы на миграцию данных и потоков аналогичны заполнению кэша L2 в обычной однопроцессорной системе.

Ключевые архитектурные особенности, которые позволяют использовать мои механизмы миграции данных и потоков, — это унифицированное представление потоков и данных с использованием возможностей и межпоточковой коммуникации и доступа к памяти через архитектурно явные очереди. Доступ к потокам и данным в моей архитектуре, ADAM, осуществляется с использованием представления возможностей с тегами, которые кодируют информацию о базе и границах. Другими словами, каждый указатель связан с областью данных, к которой он может получить доступ, и эта информация упрощает выяснение того, что перемещать во время миграции. Архитектурно явные очереди, с другой стороны, упрощают многие вспомогательные задачи, связанные с миграцией потоков и данных, такие как перемещение стеков, миграция и размещение структур связи, одновременный доступ к мигрирующим структурам и обновления указателей после миграции.

В моей диссертации также описывается схема реализации ADAM, названная «Q-Machine». Предполагается, что технология реализации — это 35 нм КМОП-кремний, доступный в больших объемах около 2010 года, и не имеющий кэшей данных; вместо этого он полагается на механизм миграции и многопоточность для поддержания хорошей производительности и высокой загрузки процессора. Предлагаемая реализация моделируется с помощью системного симулятора ADAM (ASS); именно этот симулятор предоставляет результаты, на основе которых оценивается архитектура ADAM. Обратите внимание, что для реализации ADAM не требуется передовых технологий; при желании можно реализовать ADAM уже сегодня. Технологическая точка 2010 года была выбрана для оценки архитектуры ADAM, поскольку она будет соответствовать вероятным временным рамкам реализации архитектуры.



## 1.2 Организацион этой работы

Глава 2, «Предыстория», рассматривает некоторые преимущества и недостатки схемы миграции по сравнению с более традиционными схемами управления задержкой. Она также рассматривает на высоком уровне некоторые проблемы, с которыми сталкивались предыдущие схемы миграции; более подробный обзор механизмов миграции представлен в Главе 4. Глава 2 завершается дифференциацией этой работы от ее предшественников в кратком обсуждении архитектурной родословной ADAM и ее реализации Q-Machine.

Глава 3, «Aries Decentralized Abstract Machine», подробно описывает ADAM. В этой главе закладывается основа для модели программирования ADAM с помощью простого примера кода, за которым следует обсуждение архитектурных деталей, относящихся к реализации миграции. Подробное обсуждение других архитектурных особенностей можно найти в Приложении В.

Глава 4, «Механизм миграции в децентрализованной вычислительной среде», представляет реализацию механизмов миграции. Глава начинается с обзора предыдущих работ, связанных с миграцией данных и потоков; этот обзор включает как механизмы, так и алгоритмы управления миграцией, поскольку детали их реализации тесно связаны. Затем я подробно описываю механизм миграции.

Глава 5, «Реализация ADAM: аппаратное обеспечение и моделирование», описывает реализацию ADAM. Эта реализация известна как Q-Machine. В этой главе суммируются предположения об организации машины и технологии реализации симулятора, используемые для оценки моих механизмов миграции.

В следующей главе «Характеристика машины и миграции» (глава 6) я характеризую производительность реализации. Глава начинается с двух простых микроядерных тестов и некоторого формального анализа механизма миграции. Затем я представляю результаты для некоторых более комплексных тестов, Quicksort, Matrix Multiply и N-Body, с простыми эвристиками управления миграцией, управляющими механизмами миграции.

Диссертация завершается в главе 7 обсуждением дальнейших разработок архитектуры ADAM, областей для улучшения и дальнейшего исследования, а также языков программирования для машины. Обратите внимание, что хотя подробное обсуждение языков программирования для ADAM выходит за рамки этой диссертации, я не работал в вакууме языка программирования. Сильная сторона использования абстрактной модели

машины заключается в том, что разработчики компиляторов могут начать свою работу в первый день, и на самом деле это так. Бенджамин Вандивер, студент магистра инженерных наук в моей исследовательской группе, разработал два языка, Couatl и People, и компиляторы для этих языков для архитектуры ADAM. Couatl — это базовый объектно-ориентированный язык, который мы использовали на ранних этапах разработки архитектуры, чтобы выработать абстрактную модель машины и определить уникальные сильные и слабые стороны архитектуры на основе очередей. Последующий язык, People, является более сложным языком, поддерживающим потоковые конструкции, которые используют доступность архитектурных очередей на уровне языка. Я отсылаю заинтересованных читателей к его магистерской инженерной диссертации [Van02].

Краткое изложение уровней абстракции, используемых в этой диссертации, можно найти на рисунке 1-1. ADAM является чистой абстракцией, границей между компиляторами и оборудованием. Q-Machine — это реализация ADAM, которая реализует быстрые механизмы миграции данных и потоков, которые стали возможны благодаря ADAM. ADAM System Simulator (ASS) — это моя программная симуляция Q-Machine, написанная на Java. Q-Machine также может быть реализована непосредственно в оборудовании, но это выходит за рамки данной диссертации.

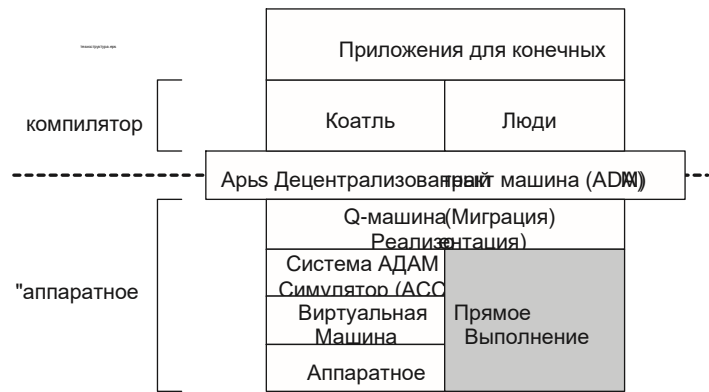


Рисунок 1-1: Обзор уровней абстракции в этой диссертации. Couatl и People — компиляторы, написанные Беном Вандивером.

Я также предоставляю набор приложений, описывающих различные технические детали архитектуры, включая детали архитектуры ADAM на уровне битов, реализацию файла физической очереди (PQF), реализацию сетевого интерфейса, сетевые протоколы и коды операций ADAM.

## Глава 2

# Фон

TSMC не видит непреодолимых препятствий на пути к масштабированию [технологии кремниевых КМОП] до 9 нм узла. Вопрос в том, будет ли рынок готов к этому?

— *Кэлвин Ченминг Ху, технический директор TSMC, на лекции в Массачусетском технологическом институте*

Эта глава начинается с характеристики архитектуры ADAM с точки зрения использования ею методов управления задержкой. Затем в этой главе более подробно обсуждается сравнение различных методов миграции. Наконец, эта глава завершается обсуждением архитектурной родословной ADAM.

## 2.1 Методы управления задержками

Многочисленные методы управления задержкой доступны компьютерным архитекторам, желающим проектировать большие параллельные машины. Методы управления задержкой можно разделить на две большие категории: сокращение задержки и сокрытие задержки.

### 2.1.1 Сокращение задержки

Методы сокращения задержки включают архитектурные компромиссы для оптимизации задержки доступа к локальной памяти, такие как неравномерный доступ к памяти (NUMA) и архитектура кэш-памяти только (COMA). Архитектуры NUMA явно справляются с пространственной реальностью больших машин; таким образом, ссылки на локальную память быстрее, чем ссылки на удаленную память. Это контрастирует с архитектурами на основе шин, которые имеют равномерное время доступа к памяти. NUMA обычно используют пространственные сети взаимосвязей, которые по своей сути более масштабируемы, чем архитектуры на основе шин. Хотя NUMA обеспечивают лучшую масштабируемость, они сталкиваются с проблемой того, как организовать данные так, чтобы была достигнута оптимальная производительность. Одним из популярных методов решения проблемы размещения данных является использование механизма когерентности кэша на основе каталогов. Примерами когерентных NUMA с кэшем (ccNUMA) являются DASH

Стэнфорда [LLG 92] и Alewife Массачусетского технологического института [ABC 95]. С другой стороны, СОМА характеризуются автоматической миграцией данных с помощью «привлекательных воспоминаний». СОМА также используют пространственные сети взаимосвязей, которые характеризуются неравномерным временем доступа к памяти, но в СОМА память не имеет домашнего местоположения. Данные мигрируют в кэш-когерентном режиме по всей машине к своим точкам доступа. СОМА имеют недостаток в виде дополнительной аппаратной сложности, но имеют преимущество перед машинами NUMA, когда рабочий набор данных больше, чем размер кэша NUMA. Архитектура ADAM похожа на архитектуру СОМА, за исключением того, что ADAM также имеет миграцию потоков и что нет кэшей — другими словами, в машине может быть только одна действительная копия фрагмента данных. Удаление семантики кэша из памяти снижает требования к оборудованию, но приводит к тому, что ADAM теряет преимущество автоматической репликации данных. ADAM пытается компенсировать эту потерю, предоставляя распознаваемый оборудованием неизменяемый тип данных, который является однократно записываемым и может свободно копироваться по всей машине. Миграция потоков также помогает компенсировать эту потерю, позволяя потокам, которые активно борются за один фрагмент памяти, мигрировать в сторону спорной памяти

расположение.

Сокращение задержки также может быть применено на более низком уровне посредством миграции, репликации, планирования, размещения и кэширования. Репликация — это свойство, присущее системам памяти с когерентным кэшем, где память может быть помечена как эксклюзивная или только для чтения, и несколько копий могут существовать по всей машине, чтобы уменьшить воспринимаемую задержку доступа на нескольких узлах. Как упоминалось ранее, ADAM обеспечивает лишь ограниченную поддержку репликации данных. Планирование и размещение — это предиктивные методы, которые пытаются сократить задержку и сбалансировать нагрузку путем выделения памяти и планирования потоков так, чтобы они находились рядом друг с другом. Планирование и размещение могут быть либо явно направлены программистом, выведены и статически связаны компилятором, либо направлены интеллектуальной системой выполнения. Планирование и размещение — важные методы сокращения задержки в любой архитектуре, но они выходят за рамки моей диссертации. Подробное обсуждение и сравнение методов миграции зарезервированы для более поздней части этой главы и главы 4.

Кэширование, пожалуй, является наиболее широко используемым механизмом сокращения задержек. Кэши сокращают задержку памяти, сохраняя последние доступные значения в быстрой памяти близко к процессору. Кэши полагаются на статистически хорошие пространственные и временные характеристики локальности доступа к данным, которые встречаются в большинстве программ. Кэши также полагаются на исключительное владение данными; поскольку копия данных создается в основной памяти, для корректного выполнения программы в среде, где возможна одновременная модификация, требуется механизм когерентности. Этот механизм когерентности может представлять проблему при масштабировании до очень больших многопроцессорных машин. В частности, простые механизмы когерентности на основе каталогов или слежки показывают плохую масштабируемость. Механизмы когерентности слежки используются в многопроцессорных системах на основе шин и страдают от ограничений пропускной способности из-за избыточного трафика когерентности по мере масштабирования систем. Протоколы на основе каталогов более масштабируемы, но у них также есть свои ограничения. При размере блока 64 байта простой протокол когерентности кэша на основе каталогов имеет накладные расходы памяти более 200% для системы с 1024 процессорами [CS99], стр. 565. Такие методы, как схемы с ограниченным указателем [ASHH88], схемы с расширенным указателем [ALKK91] и разреженные каталоги [GWM90], могут использоваться для снижения накладных расходов когерентности кэша в больших параллельных системах, но за счет более сложных протоколов или необходимости специальных механизмов для обработки крайних случаев, когда протокол дает сбой. Другая проблема с кэшами заключается в том, что масштабирование технологии не идеально; буферизованные задержки проводов растут немного быстрее, чем ожидалось, и ожидается, что ожидаемая емкость кэшей на время доступа уменьшится по мере развития технологических процессов [АНКВ00] [McF97]. Рисунок 2-1 иллюстрирует последствия неидеального масштабирования задержки проводов. Поскольку архитектура ADAM уже включает миграцию данных для сокращения задержек и может выдерживать большую задержку доступа из-за использования многопоточности и развязки, в реализации ADAM, описанной в этой диссертации, не используются кэши данных. Устранение кэшей данных снимает проблемы масштабирования кэшей данных, а также помогает снять часть давления времени доступа, вызванного технологическими ограничениями. Недостатки этого решения включают более медленное выполнение однопоточного кода и потерю автоматической репликации данных, присущей схемам

когерентности кэшей. Обратите внимание, что реализация ADAM, как упоминалось ранее, компенсирует эту потерю репликации данных частично за счет предоставления неизменяемого типа данных, а частично за счет миграции потоков в сильно оспариваемые области памяти.

## 2.1.2 Скрытие задержки

Методы сокрытия задержек включают в себя предварительную выборку, разъединение, многопоточность, ослабление согласованности памяти и инициируемую производителем связь.

Предварительная выборка — это использование предиктивных механизмов, как автоматических, так и явных, для доступа к данным до того, как вычисления потребуют данные. Эффективность предварительной выборки пропорциональна точности предиктивного механизма. Когда предиктивный механизм неверен, система может потенциально заплатить высокую цену, поскольку неправильно предиктивно выбранные данные могут вытеснить полезные данные, потребляя полосу пропускания, которая могла бы использоваться для другой полезной работы. Предварительная выборка может применяться в архитектуре ADAM, но ее реализация выходит за рамки данной диссертации.

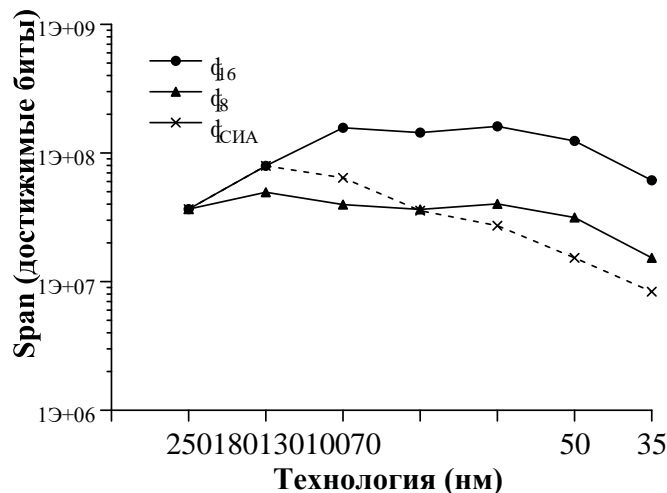


Рисунок 2-1: Доступная область чипа в верхнем уровне металла, где площадь измеряется в шеститранзисторных ячейках SRAM. Напрямую из [АНКВ00]

Разделение — это использование явных очередей для сокрытия задержек доступа или вычислений. Разделение реализовано в машинах с раздельным доступом и выполнением (DAE), таких как ZS-1 [SDV 87], архитектура WM [Wul92] и MT-DCAE [SKA01].

Разделенные архитектуры можно рассматривать как тип архитектуры программируемой предварительной выборки, хотя механизм разделения также может использоваться для разделения событий потока управления. В простой архитектуре DAE процессоры разделены на блоки доступа и выполнения, соединенные набором очередей. Блок доступа может «проскальзывать» перед блоком выполнения, эффективно выполняя предварительную выборку данных для блока выполнения. Поскольку ADAM использует явные очереди для связи с потоками и доступа к памяти, ADAM разделяет многие преимущества и проблемы архитектур DAE.

Многопоточность — это использование нескольких контекстов потоков и механизма быстрого переключения контекста для сокрытия задержек доступа к памяти. Когда один контекст потока останавливается из-за зависимости, требующей длительного доступа к памяти, другой контекст потока заменяется, тем самым поддерживая высокий уровень использования процессора. Однако многопоточность может эффективно скрыть задержку памяти только в том случае, если имеется достаточное количество готовых к запуску контекстов. По мере увеличения задержек требуется больше параллелизма. Архитектуры NEP [Smi82a] и TERA [AKK 95] применяют многопоточность для сокрытия задержек доступа; архитектура ADAM также использует эту технику.

Модели согласованности памяти с ослаблением и связь, инициированная производителем, являются архитектурными и программистскими методами сокрытия задержки. Модели согласованности памяти с ослаблением скрывают задержку, предоставляя системам большую гибкость в сокрытии задержек записи [LW95]. Выбор модели согласованности памяти оказывает большое влияние на то, как машина программируется (или компилируется).

ADAM использует слабую модель упорядочения [DS90], похожую на ту, что используется в Alpha [CS99]. Конечно, каждый поток гарантированно записывает и считывает в программном порядке в ADAM. Инициированная производителем связь уменьшает задержку, отсекая половину кругового пути, когда отношения производителя и потребителя четко определены. Вместо того, чтобы потребитель отправлял сообщение для запроса данных и ждал ответа, инициированная производителем связь помещает данные в кэш или очередь потребителя. В системе с когерентным кэшем это может привести к более высокому трафику когерентности, поскольку все общие копии должны обновляться при каждой записи [LW95]. В ADAM инициированная производителем связь является единственным режимом

связи при использовании сопоставленных очередей. Для этого стиля связи в ADAM нет накладных расходов на когерентность, поскольку пространство имен очереди отделено от пространства имен памяти, и все сопоставления очередей являются исключительными по определению.

## 2.2 Механизмы миграции

Механизмы миграции, как правило, подгоняются под конкретную архитектуру, операционную систему или приложение. В результате, особенности схем миграции одинаково разнообразны. Например, в сети рабочих станций (NOW) механизмы миграции, как правило, работают с крупнозернистыми процессами и объектами. Миграция на NOW, как правило, находится под динамическим контролем времени выполнения, а время миграции составляет порядка десятков-сотен миллисекунд. [RC96] С другой стороны, миграция вычислений на Alewife [HWW93] реализует структурированное перемещение кадра активации по всей машине с использованием статически скомпилированных директив миграции, что обеспечивает время миграции порядка нескольких сотен циклов процессора.

В наименьшем общем знаменателе каждый механизм миграции должен выполнять следующие действия: определять, что перемещать, подготавливать получателя, отправлять данные и затем обрабатывать любые пересылаемые запросы или обновления указателей. Схемы миграции потоков или процессов также должны обрабатывать проблемы планирования задач. Миграция процессов в NOW невероятно неэффективна и медленна, поскольку граница абстракции для процессов слишком высока; например, перемещение процесса влечет за собой создание виртуального адресного пространства и перемещение дескрипторов файлов. [RC96] представляет более быструю, более оптимизированную версию миграции процессов, которая снимает ограничение, что производители коммуникаций должны быть заморожены во время миграции потребителей (т. е. обеспечивает одновременную связь во время миграции), но даже в этом случае миграция процессов занимает 14 мс. [CM97] также представляет более быстрые методы для работы с обновлениями указателей после миграции с использованием явно управляемых реестров указателей. Однако проблема с явно управляемыми реестрами указателей заключается в том, что если программист забывает зарегистрировать указатель, то происходит неправильное выполнение программы. DEMOS/MP [PM83], что интересно, является многопроцессорной



операционной системой, представленной более чем за десятилетие до [RC96] или [CM97], и она имеет автоматическое обновление указателя и параллельную коммуникацию во время миграции процесса. DEMOS/MP имеет явные управляемые ОС очереди коммуникаций для межпроцессного взаимодействия; это помогает обеспечить параллельную коммуникацию во время миграции процесса и упрощает обновление указателя, поскольку менеджеру миграции не нужно делать догадки или консервативные предположения о механизме коммуникации процесса. К сожалению, документ DEMOS/MP содержит мало информации о производительности его механизма миграции процесса, поэтому его сложнее сравнивать с другими работами. Механизм миграции потоков ADAM реализует многие функции механизма миграции DEMOS/MP, за исключением более мелкой зернистости и аппаратной поддержки.

На машинах типа SMP время миграции короче благодаря более тесной интеграции сетевых интерфейсов и процессоров, как правило, более быстрым сетям взаимосвязей, более мелкой гранулярности объектов и глобально общим системным ресурсам. Например, миграция страниц в DASH занимает 2 мс (около 66 000 циклов памяти) [CDV 94]. Это не включает время, потраченное на ожидание блокировок в системе виртуальной памяти ядра; в статье указывается, что время отклика для рабочих нагрузок не было улучшено из-за этих накладных расходов. Даже если можно было бы перенести потоки в DASH, просто перебросив счетчик программ через забор на другой процессор, накладные расходы на миграцию связанного с потоком состояния процесса — стека и кучи — были бы довольно большими, поскольку необходимо переместить по крайней мере две структуры памяти, возможно, на уровне детализации страницы. Таким образом, планировщик потоков должен знать объем памяти, занимаемый задачей, и использовать планирование средств кэша для достижения хорошей производительности. [CDV 94]

Active Threads [WGQH98] вводит миграцию потоков пользовательского пространства, чтобы обойти накладные расходы на миграцию потоков ядра. Кроме того, Active Threads использует простые протоколы обмена сообщениями пользовательского пространства для связи, чтобы сократить накладные расходы на копирование сообщений и буферов в пространстве ОС. Миграция потоков пользовательского пространства сокращает задержки миграции потоков примерно до 150 с (около 16 000 циклов процессора). Computation Migration [HWW93] также выполняет миграцию потоков пользовательского пространства, но более ограничительным образом. В Computation Migration статические аннотации в

пользовательском коде заставляют поток порождать новые процедуры на удаленных узлах; также [HWW93] не указывает, что ресурсы межпоточной связи переносятся. Таким образом, один поток пробирается через машину с траекторией, которая отслеживает местоположение рабочего набора данных. Computation Migration выполняется быстро, так как для запуска нового потока на удаленном процессоре требуется всего 651 цикл. Тем не менее, разбивка затрат на Computation Migration показывает, что большое количество времени тратится на связывание процедур, создание потоков и маршалинг состояния потоков. В качестве примечания, Computation Migration не используется в качестве сравнительного бенчмарка для механизма миграции ADAM, поскольку Computation Migration реализует ограниченную версию миграции потоков, которая не обеспечивает уровень динамизма или параллелизма, найденный в следующей по скорости реализации миграции, Active Threads. Следовательно, Active Threads используется в качестве точки сравнения для механизма миграции ADAM.

Обратите внимание, что этот краткий обзор механизмов миграции подробно изложен в разделе «История вопроса» главы 4.

### 2.2.1 Обсуждение

Архитектура ADAM структурирует потоки, данные и их механизмы связи таким образом, чтобы исключить или радикально сократить накладные расходы, испытываемые механизмами миграции, описанными выше. Например, почти все механизмы миграции должны иметь дело с обновлениями указателей и пересылкой сообщений. Проблема в том, что межпоточные коммуникации почти всегда используют ресурсы памяти, поэтому любая миграция потоков требует перемещения стеков, структур ОС или структур связи, выделенных в куче. Архитектура ADAM уплотняет структуры связи в явно именованные ресурсы с помощью явных очередей. В результате состояние связи сохраняется как часть состояния потока, а миграция потока обычно включает одну операцию копирования. Использование ограниченных возможностей для представления состояния потока в памяти, а также всех структур данных кучи также упрощает миграцию, поскольку область памяти, которая должна быть скопирована во время миграции, может быть напрямую вычислена с

учетом указателя на поток или объект данных. Использование ограниченных возможностей также обеспечивает большую гибкость в выборе гранулярности миграции по сравнению со схемами, требующими миграции на уровне страниц, например, используемыми в DASH [CDV 94]. Еще одним преимуществом ограниченных возможностей является невозможность ложного совместного использования данных. Например, в обычной системе два объекта могут по случайности совместно использовать строку кэша или страницу памяти (см. рисунок 2-2). Если к двум объектам памяти одновременно обращаются потоки на разных узлах, строка кэша или страница памяти либо в конечном итоге будут перебрасываться между узлами, либо один поток будет страдать от несправедливого времени доступа. С другой стороны, ограниченные возможности не помогают, когда программист пишет код, который явно совместно использует объекты среди множества разбросанных потоков. В этом случае миграцию потоков следует использовать для минимизации задержек доступа.



Рисунок 2-2: Иллюстрация проблемы ложного обмена.

## 2.3 Архитектурная родословная

Генезис архитектуры ADAM лежит в архитектуре потока данных, архитектуре с разделенным доступом/выполнением (DAE), архитектуре «процессор в памяти» (PIM) и многопроцессорной архитектуре (CMP), а также архитектуре кэш-памяти (COMA).

### 2.3.1 Поток данных

ADAM, пожалуй, наиболее тесно связан с семейством архитектур потоков данных, в частности, с \*T.

Поэтому в настоящее время важно тщательно изучить машины, обрабатывающие потоки данных.

Машины потока данных являются прямой реализацией графов потока данных в вычислительном оборудовании. Дуги на графе потока данных разлагаются на токены. Каждый токен является продолжением; он содержит набор инструкций и свой контекст оценки. Длина выполнения инструкции и метод контекста оценки, инкапсулированные в токен, могут характеризовать спектр архитектур потоков данных. В архитектуре потока данных с тегированными токенами MIT (TTDA) каждый токен представляет примерно одну инструкцию и ее непосредственные зависимости и результаты, а хранение токенов управляется неявно. TTDA развилась в архитектуру Monsoon, которая имеет явное управление контекстом оценки и токены с одной инструкцией. С Monsoon токены содержали значение; указатели на инструкцию и указатели на контексты оценки, которые являются сгенерированными компилятором выделениями кадров в линейно адресуемой структуре. Monsoon развился в архитектуры P-RISC и \*T, которые представляют собой машины с токенами, которые эффективно ссылаются на трассировки инструкций и относительно большие явно выделенные кадры в стиле «стекового кадра». Токены в P-RISC и \*T несли только указатель инструкции и указатель кадра, в отличие от любых фактических данных [AB93] [NA89]. Можно было бы пойти еще дальше и заявить, что архитектура одновременной многопоточности (SMT) представляет собой машину потока данных с таким же количеством токенов, сколько имеется контекстов потоков, и что традиционная архитектура фон Неймана представляет собой машину потока данных с одним токеном. [LN94] дает прекрасный обзор машин потока данных и анализ их недостатков.

Машины потока данных, хотя и элегантны, имеют несколько фатальных недостатков. Их эволюция от TTDA до почти RISC-архитектур дает подсказку о том, в чем заключаются эти недостатки. Довольно абстрактная TTDA разложила графы потока данных до почти атомарного уровня инструкций. Тысячи токенов создаются в ходе даже простого выполнения программы, поскольку токены могут быть сформированы и отправлены до разрешения зависимостей. [AB93] утверждает, что «эти токены представляют данные, локальные для неактивных функций, которые ожидают возврата значений, проходящих вычисление в других функциях, вызываемых из их тел». Выполнение любого токена требовало ассоциативного поиска по пространству всех токенов для токенов, которые содержали результаты, удовлетворяющие зависимостям данных текущего токена.

Ассоциативная структура из нескольких тысяч элементов, необходимая для выполнения этого поиска, нереализуема даже после двадцати лет масштабирования процесса.

Еще один недостаток ранних машин Dataflow заключается в том, что каждый токен представляет собой событие синхронизации с высокими накладными расходами. [Jan88] указывает, что архитектуры фон Неймана также выполняют событие синхронизации между каждой инструкцией, но метод синхронизации очень легкий:  $IP = IP + 1$  или  $IP = \text{цель ветвления}$ . Это позволяет архитектурам фон Неймана очень быстро проходить через линейный код. К счастью для сторонников фон Неймана, большую часть кода, написанного на сегодняшний день, можно достаточно выпрямить с помощью либо прогнозирования ветвлений, либо планирования трассировки, чтобы получить хорошую производительность такой системы. P-RISC и \*T в некоторой степени использовали эту силу архитектур фон Неймана, позволяя токenu представлять то, что по сути является трассировкой выполнения и стековым фреймом. \*T на самом деле имеет очень похожую на ADAM одноузловую архитектуру: она делит один узел на сопроцессор синхронизации и процессор данных. Процессор синхронизации отвечает за планирование потоков и решение проблем синхронизации, в то время как исключительная задача процессора данных заключается в эффективном выполнении линейного кода. Однако на этом сходство заканчивается, поскольку архитектура \*T в первую очередь фокусируется на сокрытии задержки посредством быстрого и эффективного планирования потоков, запуска и переключения контекста. Хотя сокрытие задержки посредством многопоточности является важной частью архитектуры ADAM, также очень важно сократить задержку, предоставив механизмы для эффективной миграции данных и потоков между узлами процессора. Общая организация ADAM отражает это внимание к механизмам миграции. Кроме того, тщательное изучение стратегии реализации, изложенной в [PBB93], выявляет ряд важных различий (и сходств) между ADAM и \*T. Одним из существенных отличий является использование ADAM интерфейса на основе очередей между потоками с неявной синхронизацией через пустые/полные биты, аналогично схеме, используемой в MMachine [FKD 95]. \*T использует интерфейс на основе регистров с кэшем микропотоков для обеспечения эффективного переключения контекста и явной обработки на уровне программы сообщений, которые не удалось внедрить в сеть. Использование самосинхронизирующихся очередей непрозрачной глубины в ADAM помогает смягчить перегрузку сети и сбои в планировании.

### 2.3.2Разъединенный-Доступ/Выполнение

Машины с раздельным доступом/выполнением (DAE) — это одноузловые процессоры с отдельными механизмами выполнения и доступа. Эти механизмы связаны с архитектурно видимыми очередями, которые используются для сокрытия задержек доступа к памяти. Код для этих машин обычно вручную или компилятором разбивается на поток доступа и выполнения; задержки скрыты, поскольку поток доступа, который обрабатывает запросы памяти, может «проскальзывать» впереди потока выполнения. Было построено относительно немного машин, которые явно поддерживают DAE. Архитектура была впервые предложена в [Smi82b] и позже реализована как Astronautics ZS-1 [SDV 87]. [MSAD90] подробно характеризует производительность сокрытия задержек ZS-1, а [MSAD91] сравнивает производительность ZS-1 с IBM RS/6000. Сравнение DAE и суперскалярных архитектур можно найти в [FNN93], а сравнение DAE и VLIW архитектур можно найти в [LJ90]. Другая предложенная архитектура DAE — это архитектура WM [Wul92], а новый поворот в архитектурах DAE, где блок доступа фактически совмещен с памятью, предлагается в [VG98]. Архитектура, описанная в этой работе, параллельна многим идеям в [VG98].

Основная идея, содержащаяся во всех ранее цитированных работах, заключается в том, что путем разумного разделения процессора на два пространственно распределенных процессора можно добиться более чем двукратного прироста производительности. Это сверхлинейное ускорение является результатом задержки, которая была архитектурно обойдена либо путем разрешения подсистеме памяти эффективно проскальзывать вперед и предварительно загружать данные в блок выполнения, либо путем физического размещения блока доступа с памятью. Идеи DAE на самом деле могут быть применены в общем виде к любой машине с большим объемом явного параллелизма, просто разделив каждую программу на два потока, поток доступа и поток выполнения. Преимущество явных машин DAE заключается в том, что синхронизация между потоками доступа и выполнения происходит очень быстро, поскольку они связаны через аппаратные очереди, в отличие от программно эмулируемых очередей. Некоторые обычные машины с неупорядоченным выполнением также обеспечивают определенную степень неявного разделения доступа/выполнения через глубокие спекулятивные буферы хранения и загрузки. Однако, в целом, обычные архитектуры, которые программно эмулируют эти очереди, платят высокую цену за накладные расходы на синхронизацию. Реализации программного обеспечения,

использующие опрос для проверки пустых битов, оплачивают накладные расходы опроса плюс любое время, потерянное между фактическим событием доступности данных и событием опроса. Реализации, управляемые прерываниями, также дороги, поскольку типичные механизмы прерываний требуют вмешательства ядра.

Другое важное сообщение заключается в том, что очереди подобны обходным конденсаторам для компьютерных архитектур. Очереди фильтруют низкие частоты неравномерных моделей доступа высокопроизводительного кода и помогают отделить сторону спроса вычисления от стороны предложения вычисления. Как и обходные конденсаторы, постоянная времени очереди (т. е. размер очереди) должна быть достаточно большой, чтобы отфильтровать средний всплеск, но не настолько большой, чтобы уменьшить доступную полосу пропускания сигнала и затруднить выполнение важных задач, таких как переключение контекста. Накладные расходы структуры очереди также должны быть небольшими, чтобы можно было реализовать преимущества очередей.

К сожалению, простые машины DAE в целом страдают от нескольких проблем. Нет компиляторов, которые генерируют явный доступ и потоки кода выполнения; большинство тестов и симуляций в цитируемых работах были с циклами доступа и выполнения, закодированными вручную. Кроме того, эффективность DAE сомнительна для сложных циклов и программ со сложными и/или динамическими графами потоков данных. Простой DAE нацелен на сокрытие задержек памяти и не более того. Однако основная идея разделения блоков доступа и выполнения является мощной; особенно если физические блоки доступа и выполнения разрешено динамически назначать одному виртуальному потоку управления, как в случае с ADAM. Создание этих «виртуальных» машин DAE позволяет блокам доступа и выполнения мигрировать по всей машине и оптимизировать задержку на основе потока за потоком. Достаточно гибкая инфраструктура также позволила бы объединить несколько блоков выполнения вместе, тем самым обеспечивая своего рода развертывание цикла и возможность потоковых вычислений без какой-либо модификации кода. Поскольку эта цепочка является динамической, такую машину можно модернизировать, чтобы она имела больше процессоров, и более высокая производительность могла бы быть реализована без перекомпиляции кода. Эта идея виртуальной архитектуры DAE является важной частью архитектуры ADAM.

### 2.3.3 Процессор в памяти (PIM) и чип-мультипроцессоры (CMP)

Недавние достижения в технологии процесса сделали возможным интеграцию достаточного количества SRAM на кристалле для создания автономного процессорного узла с одним кристаллом. Кроме того, наличие DRAM, встроенной в тот же кристалл, что и процессор, открывает дверь к еще более высоким уровням интеграции памяти [Corb] [Mac00] [Cora]. Такая интеграция процессоров и памяти на одном кристалле называется «процессор в памяти» (PIM). Тот факт, что память включена в тот же кристалл, что и процессор, подразумевает преимущество в мощности и производительности из-за устранения емкостей проводки чип-чип и длин проводов. Это также обеспечивает преимущество в производительности, поскольку между банком памяти и процессором можно проложить больше проводов, чем в дискретном решении процессор-память. По мере совершенствования технологии процесса станет возможным разместить несколько ядер процессора и память на одном кремниевом кристалле. Этот стиль реализации известен как Chip Multi-Processor (CMP). Статья, в которой суммируются некоторые ключевые аргументы в пользу архитектур CMP, находится в [ONH 96]. Некоторые предложенные архитектуры, которые используют преимущества некоторой комбинации технологии встроенной памяти и технологии многопроцессорности чипа, включают RAW [LBF 98], I-RAM [KPP 97], активные страницы [OCS98], DRAM с разделенным доступом [VG98], Terasys [GHI94], SPACERAM [Mar00] и Hamal [Gro01].

Уровень производительности, доступный пользователям встроенной DRAM, замечателен. Традиционно, DRAM считается медлительным танкером памяти, в то время как SRAM — король скорости. Недавнее ядро DRAM, представленное MoSys (так называемое 1-T SRAM), доступное на процессе TSMC, доказало, что DRAM имеет место в высокопроизводительных архитектурах [Cora]. 1-T SRAM основано на технологии DRAM, но имеет необновляемый интерфейс, как SSRAM (синхронная SRAM). Производительность этого макроса также достаточно высока — 2-3 цикла времени доступа при 450 МГц в процессе 0,13 мкм — чтобы полностью исключить необходимость в кэшах данных в конструкции процессора. Обратите внимание, что целевые частоты процессора для ADAM находятся на одном уровне с скомпилированными целевыми частотами процессора «soft core», которые обычно в 2-4 раза



ниже уровня полностью настраиваемых процессоров, разработанных Intel, AMD и Compaq. Предполагается, что ADAM будет реализован с использованием портативного RTL поток проектирования, оптимизированный для быстрых циклов проектирования и переносимости на новейшие технологические процессы, предлагаемые литейными заводами. Сокращенное время внедрения и архитектура CMP ADAM помогают компенсировать потерю производительности при использовании скомпилированного потока проектирования. Наконец, поскольку 1T SRAM имеет структуру ячеек памяти DRAM, плотность этих макросов аналогична встроенным макросам DRAM, предлагаемым в других процессах (2,09 мм на Мбит для макроса DRAM в процессе Cu-11 IBM [Mac00] против 1,9 мм на Мбит для макроса MoSys в TSMC 0,13 м логический процесс [Кора]).

Архитектура ADAM использует как высокий уровень логической интеграции, доступный в будущих технологических процессах, так и доступность готовых, быстрых и плотных запоминающих устройств для создания распределенной массивно-параллельной архитектуры с хорошей производительностью однопоточного кода.

#### **2.3.4 Архитектуры кэш-памяти**

Хотя архитектура, предложенная в этой диссертации, не имеет кэшей данных, можно утверждать, что скорость памяти, используемой в узлах процессора, позволяет отнести их к программно-управляемым кэшам. Следовательно, важно рассмотреть класс машин, известных как архитектуры Cache Only Memory (COMA).

Наиболее релевантная машина в этом классе — Data Diffusion Machine (DDM) [MSW93]. DDM полагается на миграцию данных через неявную семантику кэшей. Поскольку эта работа так тесно связана с миграцией данных и ее контролем, подробное обсуждение того, как ADAM относится к

DDM откладывается до раздела 4.

## Глава 3

# Овен Децентрализованная Абстрактная Машина

В то время как Ньютон должен был сказать (саркастически, на самом деле, но это уже другая история), что он видел дальше, стоя на плечах гигантов, большинство из нас приседает на коленные чашечки пигмеев. Но это сказано в самом приятном смысле.

— *Томас Х. Ли, заявление члена комиссии ISSCC 2002 г.*

Aries Decentralized Abstract Machine (ADAM) — это абстрактная параллельная компьютерная архитектура, оптимизированная, помимо прочего, для быстрой миграции данных и потоков. В этой главе представлен обзор архитектуры с выделением основных особенностей, которые позволяют реализовать высокопроизводительную миграцию. Сначала представлен простой пример кода, чтобы познакомить читателей с основными абстракциями связи и памяти ADAM. За примером следует более формальное, глубокое обсуждение различных особенностей ADAM.

### 3.1 Введение в ADAM на примере кода

ADAM имеет мелкозернистую многопоточную модель программирования. Межпотокное взаимодействие и доступ к памяти осуществляются через явные ресурсы очереди. Кроме того, память абстрактна; указатели представлены как возможности с базовыми и связанными тегами. Программисты не могут создавать возможности; они должны запрашивать их у машины через код операции ALLOCATE.

### 3.1.1 Основы

Простой пример программы, иллюстрирующий основные черты архитектуры, можно увидеть на рисунке 3-2. Этот код иллюстрирует связывание процедур, распределение возможностей и сопоставление памяти.

```
MOVES 2, д0          ; инициализируем q0 числом 2
MOVES 1, кв.1
MOVES 4, кв.1         ; инициализируем q1 числами 1 и 4
ДОБАВЛЯТЬ @q0, q1, q2
    ; в этой точке q2 имеет 3, q0 имеет 2, q1 имеет 4
ДОБАВЛЯТЬ q0, q1, @q2
    ; в этот момент q0 пуст, q1 пуст, а q2 имеет 6
```

---

Рисунок 3-1: Демонстрация модификатора копирования/удаления (@).

Базовый формат ассемблерных опкодов — ОР qa,qb,qc, где ОР — операция, qa и qb — аргументы, а qc — результат. Каждая операция может иметь ноль, один или два аргумента, а один из аргументов может быть константой. Существуют также некоторые важные опкоды, которые не следуют этому формату, например MAPQC, который будет обсуждаться вскоре. Также обратите внимание, что каждый спецификатор очереди может быть изменен с помощью модификатора @ (копировать/затирать). Рисунок 3-1 демонстрирует работу модификатора @. При чтении @ указывает, что инструкция должна скопировать значение из очереди аргументов, а не вывести его из очереди. При записи @ указывает, что инструкция должна перезаписать («затереть») самое новое значение в очереди, если оно есть, вместо того, чтобы ставить значение в очередь. Если очередь назначения пуста, оператор @ не имеет никакого эффекта. Оператор @ удобен при работе с временными объектами, которые часто используются повторно; Без него всякий раз, когда результат используется более одного раза, программисту или компилятору пришлось бы включать специальную инструкцию для дублирования значений.

### 3.1.2 Вызов конвенции

В ADAM соглашение о вызовах заключается в том, что каждая процедура — это новый поток. Аргументы и возвращаемые значения передаются через сопоставления очередей. Код на рисунке 3-2 демонстрирует это соглашение о вызовах.

Вызывающий, main, вызывает testStubby, выполняя инструкцию SPAWNC q2,testStub,q0. Эта инструкция запускает новый поток с его счетчиком программ, установленным на метку testStub, и возвращает контекстный идентификатор нового потока в q0. Аргумент q2 — это метрика порождения; это позволяет программисту контролировать размещение новых потоков. В этом случае метрика порождения была инициализирована значением 1, что приводит к запуску нового потока на некотором узле, находящемся на один сетевой переход.

После создания нового потока вызывающий сопоставляет очередь с пространством очереди вызываемого, чтобы инициировать передачу аргументов. Сопоставление очереди приводит к тому, что значения, записанные в отображенную очередь, в конечном итоге появляются в цели карты. Место хранения данных, записанных в отображенную очередь, — это цель карты.

Кроме того, связь через карты очередей является только push-технологией; нельзя читать из отображенной очереди. Следовательно,

основной:

```

        MOVECC1, д2          ; установить метрику появления на 1
        SPAWNC q2, testStub, q0 ; создать удаленный поток
        MAPQC q1, q0, @q0      ; карта для моего ребенка
        ПРОЦИД q1              ; отправить мой procID дочернему
                               процессу
    ДВИГАТЬСЯ к20, к22          ; ожидание возвращаемого значения от
                               дочернего элемента
        ММЛ к40, к41           ; объявить q40, q41 как очереди
                               загрузки
    ДВИГАТЬСЯ @q22, q40         ; инициализировать q40 с
                               возможностью
        MOVECLO, q40           ; извлечь данные со смещения 0
        ПЕЧАТЬQ q41            ; распечатать его (инструкция для
        ОСТАНОВИТЬ             конкретной сим-карты)
testStub:
    ДВИГАТЬСЯ q0, q100         ; сохранить вызывающую функцию в
                               q100
    MAPQC q1, q20, @q100 ; мой q1 -> q20 моего вызывающего
    абонента
    MOVECC0, д2               ; установить метрику выделения на 0
    АЛЛОКАТЕК q2, 8, q10      ; выделить 8-словную локальную
                               возможность
    ММС к30, к31              ; объявить q30, q31 как очереди
                               хранения
    ДВИГАТЬСЯ@q10, q30         ; init q30 с возможностью
    MOVECLO, q30               ; сохранить данные 10 по смещению 0
    MOVECL 10, q31
    МСИНК                     ; убедитесь, что магазин взял на себя
                               обязательство
    ДВИГАТЬСЯ@q10, q1         ; отправить возможность моему
    ОСТАНОВИТЬ                вызывающему абоненту

```

---

Рисунок 3-2: Простой пример кода, демонстрирующий связывание процедур, порождение потоков, выделение памяти и доступ к памяти.

После сопоставления очереди она становится доступной только для записи; чтение из сопоставленной очереди приводит к неопределенному поведению. В этом примере новый поток ожидает все свои аргументы в  $q_0$ , поэтому вызывающий объект сопоставляется с новым потоком с помощью инструкции `MAPQC q1,q0,@q0`. Обратите внимание, что инструкция `MAPQC` имеет необычную семантику. Первые два аргумента на самом деле являются непосредственными константами; другими словами, они интерпретируются просто как номера очередей, а не как источники для операндов. Первое значение,  $q_1$ , указывает локальную очередь для сопоставления. Второе значение,  $q_0$ , указывает номер очереди цели карты. Последний аргумент,  $@q_0$ , указывает очередь, из которой следует считывать идентификатор контекста цели карты. Я выбрал первые два значения в качестве постоянных значений, поскольку программисты или компиляторы обычно точно знают, какими должны быть номера исходной и целевой очередей сопоставления.

Теперь, когда вызывающий объект сопоставил очередь аргументов с вызываемым объектом, вызывающий объект сначала передает свой идентификатор контекста вызываемому объекту. Получив идентификатор контекста вызывающего объекта, вызываемый объект сопоставляет очередь возврата вызывающему объекту. В этом примере вызывающий объект и вызываемый объект договариваются по соглашению, что  $q_{20}$  является очередью возвращаемых значений. Рисунок 3-3 иллюстрирует состояние вызывающего объекта и вызываемого объекта после настройки очередей аргументов и возврата.

### 3.1.3 Распределение памяти и доступ к ней

Следующий набор инструкций в нашем примере кода демонстрирует распределение памяти и доступ к ней. Распределение памяти в ADAM выполняется с помощью инструкции `ALLOCATE`, а доступ к памяти осуществляется посредством сопоставления очередей.

В этом конкретном примере инструкция `ALLOCATEC q2,8,q10` используется для создания новой возможности.  $q_2$  — это метрика выделения, аналогичная метрике порождения, используемой кодом операции `SPAWNC`. В этом случае  $q_2$  инициализируется значением 0, поэтому эта инструкция запрашивает выделение локальной памяти.

Следующая инструкция, MMS q30,q31, объявляет q30 и q31 очередями хранения. Аргументы MMS являются непосредственными константами, аналогично инструкции MAPQC. После инструкции MMS q30 является очередью адресов хранения, а q31 является очередью данных хранения. Данные могут быть сохранены в памяти с использованием этой пары отображений очередей путем помещения пар адресов и данных в соответствующие очереди. Перед сохранением данных с использованием этих очередей очередь адресов хранения должна быть инициализирована с возможностью хранения. Это выполняется инструкцией MOVE @q10,q30; она копирует выделенную возможность в q10 в очередь адресов хранения q30. Последующие записи в очередь адресов хранения должны быть постоянными смещениями относительно начальной возможности; подсистема памяти отвечает за добавление этого смещения и проверку на предмет нарушений границ. Запись другой возможности в очередь адресов хранения приводит к повторной инициализации очереди адресов хранения с новой возможностью.

(заглос на пустом q20)

```

основной: MOVECC 1, q2
SPAWN q2, testStub, q0
MAPQC q1, q0, q0
ПРОЦД q1
ПЕРЕМЕЩЕНИЕ q20, q22ПК
МММ q10, q41
ПЕРЕМЕЩЕНИЕ @q22, q40
MOVECL0, q40
ПЕЧАТЬQ q41
ОСТАНОВИТЬ

testStub: ПЕРЕМЕЩЕНИЕ q0, q100
MAPQC q1, q20, @q100
MOVECC0, д2ПК
АЛЛОКАТЕК q2, 8, q10
ММС q30, q31
ДВИГАТЬСЯ @q10, q30
MOVECL0, q30
MOVECL 10, q31
ДВИГАТЬСЯ @q10, q1
МСИНК
ОСТАНОВИТЬ

```

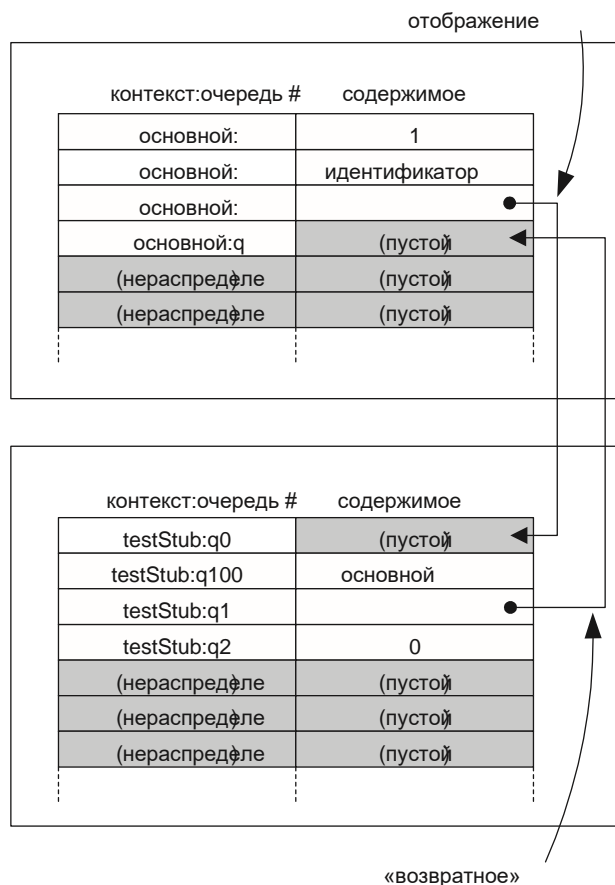
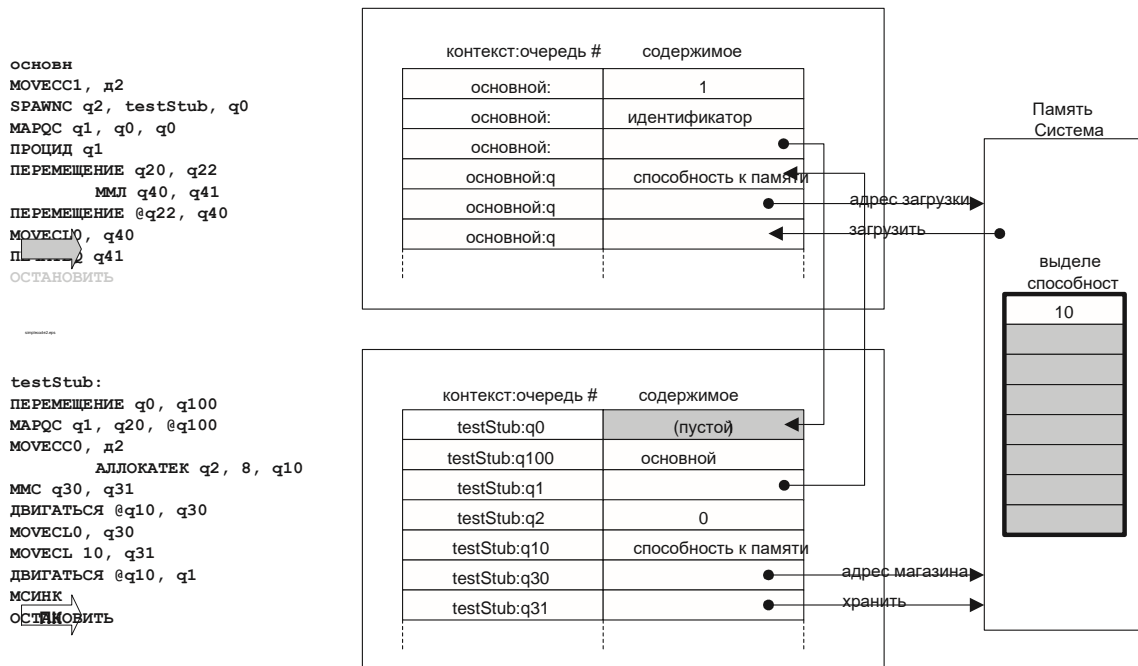


Рисунок 3-3: Состояния потока после создания потока и связывания процедур.

В нашем примере кода одно значение, 10, хранится по смещению 0. Затем поток testStub выполняет MSYNC, чтобы убедиться, что хранилище зафиксировано, и отправляет возможность памяти вызывающему потоку и останавливается. Затем вызывающий, main, устанавливает адрес загрузки и очереди данных загрузки с помощью инструкции MML q40,q41. Затем main получает доступ к возможности возвращенных данных, отправляя копию возможности в очередь адреса загрузки, q40. Затем main печатает возвращаемое значение из памяти и останавливается. Инструкция PRINTQ — это удобная инструкция, используемая только в реализации симулятора для отладки. Конечное состояние нашей



машины в конце выполнения нашего примера кода показано на рисунке 3-4.

Рисунок 3-4: Состояния потоков после выделения памяти и доступа к ней.

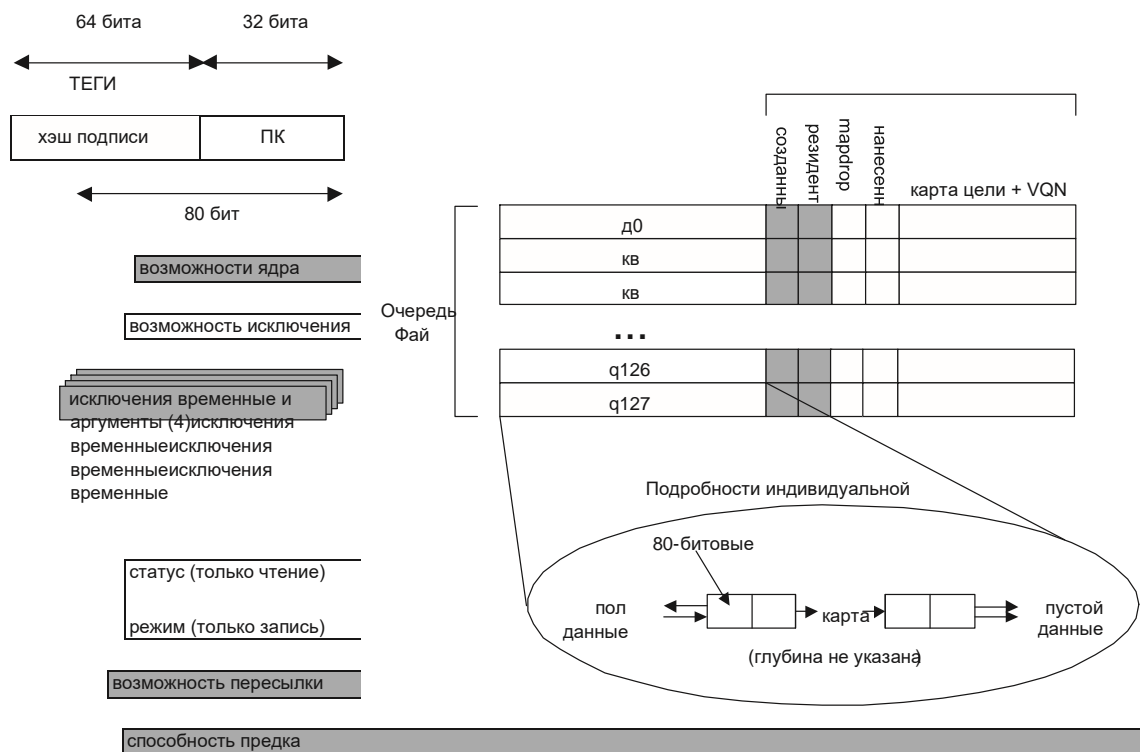
## 3.2 Модель программирования

В этом разделе подробно излагаются некоторые из основных архитектурных особенностей ADAM, представленных в простом примере кода. Для обсуждения архитектурных особенностей и деталей реализации, не имеющих прямого отношения к миграции, см. приложение В. В приложении В обсуждаются форматы инструкций, подробные разбивки

битовых полей формата возможностей, обработка исключений и взаимодействие ядра и ОС. Для всестороннего обзора кодов операций, представленных в ADAM, см. приложение D.

### 3.2.1 Темы

Основная единица вычислений в ADAM — поток. Потоки очень легковесны в ADAM и представляют собой непрозрачные, монолитные структуры памяти. Их можно было бы назвать продолжениями, за исключением того, что они несут в себе данные кадра активации в дополнение к счетчику программ и указателю среды. Состояние каждого потока имеет однозначное соответствие с областью памяти, как было показано ранее в файле именованного регистра состояний [ND91]. Адрес и границы этой области памяти идентифицируются возможностью; эта возможность называется идентификатором контекста потока. Таким образом, любой поток может быть глобально уникально идентифицирован по его идентификатору контекста, поскольку идентификатор контекста — это всего лишь указатель на память. Кроме того, количество потоков на процессор ограничено только объемом доступной памяти. Корреляция каждого состояния потока с областью памяти позволяет реализациям миграции потоков и данных использовать один и тот же базовый механизм. Сводка состояния, связанного с одним потоком ADAM, представлена на рисунке 3-5.





идентификатор контекста (возможность)

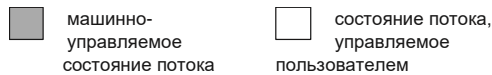


Рисунок 3-5: Модель программирования ADAM

Вместо регистров в типичной машине ADAM предоставляет очереди неопределенной глубины. Выход любой очереди может быть переназначен на вход другой очереди в другом контексте потока для межпоточковой коммуникации. Этот метод называется отображением очереди.

Аргументы и возвращаемые значения передаются между потоками через отображения очередей; в ADAM нет стека. Кроме того, связь с памятью реализуется с помощью отображений очередей. Следовательно, вся видимость в потоке и из него происходит через набор отображений очередей. Эта идея проиллюстрирована на рисунке 3-6. Использование отображений очередей упрощает реализацию миграции потоков, во-первых, изолируя все состояния потоков, включая состояние связи, в пределах одной непрерывной области памяти, а во-вторых, включая простые механизмы для управления пересылкой сообщений одновременно с миграцией. Эти механизмы миграции будут описаны в главе 4.

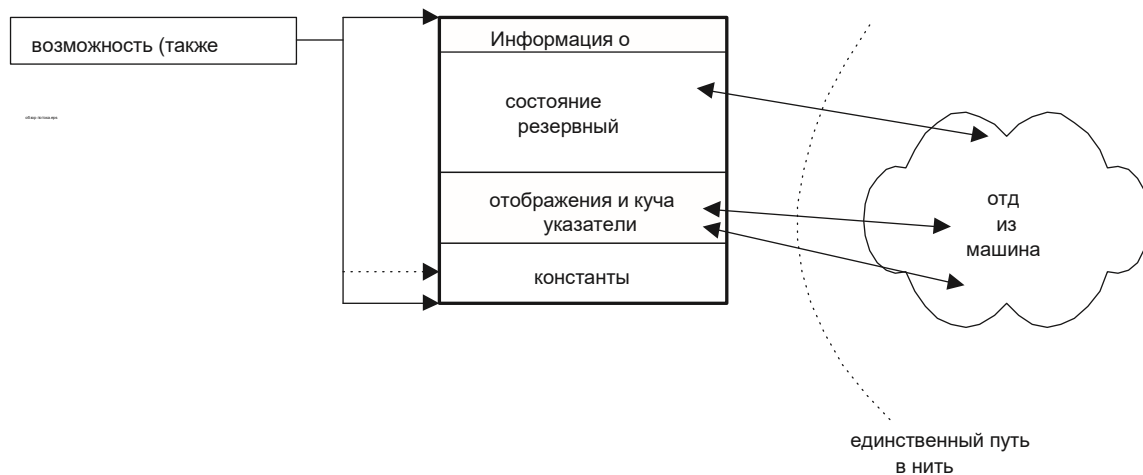


Рисунок 3-6: Структура потока ADAM

### 3.2.2 Очереди и сопоставления очередей

В первом приближении очереди, предоставляемые ADAM, имеют бесконечную глубину. Однако в реалистичной реализации производительность очередей снижается по мере того,

как в них помещается больше данных. Следовательно, хотя абстракция программирования позволяет программистам хранить большие объемы данных в очередях, этого следует избегать по соображениям производительности. Если программист соблюдает это ограничение, очереди должны работать сопоставимо с регистром в стандартной машине RISC (подробности реализации см. в приложении С). Кроме того, когда очереди используются в качестве элемента связи между потоковыми потоками, управление потоком осуществляется путем применения обратного давления (т. е. остановки очереди), пропорционального их заполнению. Это позволяет программистам объединять потоки, которые вычисляются с разной скоростью, без необходимости явно иметь дело с управлением потоком.

Отображение очереди является рекомендуемым методом для межпоточной коммуникации. Данные из любого заданного источника гарантированно придут в порядке очереди целевого контекста; однако, когда более одного отправителя сопоставлены с одним получателем, нет гарантии относительно порядка полученных значений между двумя отправителями. Узел может запросить, чтобы исходный идентификатор входящих данных был поставлен в очередь во вторичной очереди в соответствии с первичной очередью назначения, так что неоднозначность, создаваемая такой ситуацией, может быть разрешена пользовательским кодом. Хотя программист может передавать данные между потоками, передавая структуры данных, выделенные в куче, это не рекомендуется, поскольку модель памяти ADAM использует слабое упорядочение [LW95] и не дает никаких гарантий относительного порядка запросов памяти между потоками. Использование структур данных, выделенных в куче, для межпоточной коммуникации также может быть менее эффективным, чем прямые отображения очередей при наличии миграции потоков, поскольку структуры связи, выделенные в куче, не мигрируют автоматически с потоками.

При необходимости очереди ADAM могут использовать семантику регистра с помощью модификатора копирования/затирания, как описано в примере кода в начале этой главы.

### **3.2.3 Модель памяти**

ADAM использует виртуально адресуемую модель памяти на основе возможностей. Как упоминалось ранее, формат возможностей, используемый в ADAM, также кодирует базовую и связанную информацию в тегах указателя. Этот метод уже встречался ранее в [CKD94] и был улучшен в [BGKH00]. Возможности — это помеченные указатели, которые

оборудование распознает и обрабатывает иначе, чем обычные данные. В частности, обычные пользователи не могут создавать возможности самостоятельно; они должны запрашивать возможности у операционной системы или какого-либо другого доверенного механизма надзора. Эта функция помогает сделать систему более защищенной от вредоносного или неисправного кода. В случае ADAM формат возможностей дополняется битами тегов. Эти биты тегов кодируют информацию о возможности, такую как разрешения на чтение/запись и базовую/связанную информацию. Информация о базовых и связанных тегах особенно важна для реализации механизмов быстрой миграции. Учитывая возможность ADAM, можно вывести точную область данных для копирования из базовых и связанных тегов; обратите внимание, что базовый адрес, предоставленный пользователю в возможности, может отличаться от абсолютного начального адреса возможности. Кроме того, теги включают бит «только приращение». Когда этот бит установлен, пользователи могут ссылаться только на смещения к базе возможностей, которые являются положительными целыми числами, включая ноль. Это позволяет системе скрывать от пользователей информацию в верхней части каждой возможности, между абсолютным началом возможности и базовым адресом пользователя. Эта функция используется в моей реализации миграции для связывания указателя удаленного локатора данных с каждой возможностью. Функция указателя удаленного локатора данных подробно описана в главе 4. Для получения дополнительной информации о реализации кодирования базы и границ в ADAM читатели могут обратиться к приложению В.

Память распределяется по машине с использованием явного идентификатора узла как части адреса. Поле идентификатора узла и поле адреса могут красть биты друг у друга в зависимости от параметров реализации. Этот вид кодирования местоположения узла в адресе уже встречался в Cray T3E [Sco96]. Фактическая трансляция виртуальных адресов и механизмов подкачки прозрачны для спецификации и зависят от реализации. Краткое изложение формата возможностей можно увидеть на рисунке 3-7.



Рисунок 3-7: Высокоуровневая разбивка формата возможностей ADAM. Подробная разбивка на уровне битов каждого поля приведена в приложении В.

### 3.2.4 Взаимодействие с памятью

Как упоминалось ранее, в спецификации ADAM нет инструкций загрузки или сохранения; память является непрозрачным объектом, доступ к которому осуществляется только через сопоставления очередей. Коды операций MML и MMS используются для определения пар очередей загрузки и сохранения соответственно. MML принимает исходящую очередь адресов и очередь возврата данных в качестве аргументов; MMS принимает исходящую очередь адресов и исходящую очередь данных в качестве аргументов. Порядок данных в любом заданном сопоставлении очереди загрузки или сохранения в потоке гарантированно сохраняется, поскольку значения адреса и данных отправляются в подсистему памяти в режиме блокировки. Однако порядок между несколькими наборами сопоставлений не гарантируется между инструкциями MSYNC. Следовательно, доступ к одному фрагменту памяти через несколько сопоставлений очередей не рекомендуется, поскольку это может привести к недетерминированному поведению.

Блокировки и семафоры в памяти могут быть реализованы с помощью кода операции EXCH. Код операции EXCH объявляет набор из трех очередей как кортеж обмена. Одна очередь используется для указания адреса обмена, другая очередь используется как источник исходящих данных обмена, а последняя очередь используется для указания точки возврата для обмененных данных. Аппаратное обеспечение подсистемы памяти гарантирует, что этот обмен будет атомарным. Время обмена не является детерминированным: фактический обмен в ячейке памяти происходит всякий раз, когда запрос на обмен поступает в ячейку памяти назначения.

При инициализации отображения очереди памяти первый фрагмент данных, записанный в очередь адресов, должен быть возможностью, иначе будет сгенерировано исключение доступа к памяти. Последующие обращения к очереди адресов могут передавать больше возможностей или любой целочисленный тип данных. Когда целочисленный тип данных помещается в очередь памяти, предполагается, что это смещение самой последней возможности, переданной в очередь адресов. Помещение упакованного целого числа в очередь адресов приводит к возврату данных для каждого из упакованных подзначений, начиная с наименее значимого значения и заканчивая наиболее значимым значением.

Особенностью формы доступа к очереди памяти является то, что архитекторы и разработчики могут расширить спецификацию ADAM, добавив интеллект в систему

памяти. Возможности и смещения помещаются в очередь памяти, и система памяти может делать то, что ей нравится, прежде чем вернуть некоторые данные. Таким образом, система памяти может быть расширена, чтобы быть чем-то большим, чем просто таблица сохраненных значений; ее можно настроить для выполнения вычислений или автоматического обхода структур данных.

## Глава 4

# Механизм миграции в Децентрализованная вычислительная среда

Память — как оргазм. Гораздо лучше, если не нужно притворяться.

—Сеймур Крей о виртуальной памяти

### 4.1 Вступление и индукция

Идея перемещения кода и данных так, чтобы они были физически ближе друг к другу, привлекательна в любой компьютерной системе с высокими задержками связи. К сожалению, миграция создает множество новых проблем. Прежде всего, миграция потребляет вычислительные ресурсы, и системным архитекторам приходится мириться с тем фактом, что любое перемещение данных должно в конечном итоге компенсироваться за счет сокращения задержки связи. Накладные расходы механизма миграции включают не только время копирования данных, но и время, необходимое для согласования с местом назначения миграции; потенциальную задержку доступа к данным во время интервала миграции; время, необходимое для обновления любых указателей в перенесенной памяти; и любое сопутствующее влияние на использование сети и ЦП. Этот перечень ловушек производительности очень затрудняет внедрение эффективного механизма миграции в существующую архитектуру, которая была разработана без каких-либо размышлений о проблеме. Таким образом, даже если миграция данных и потоков в принципе кажется хорошей идеей, ее реализация может оказаться сложной задачей.

Архитектура ADAM и ее соответствующая реализация радикально сокращают накладные расходы, необходимые для миграции данных и потоков по сравнению с

традиционными архитектурами. Механизмы миграции данных и потоков ADAM в основном идентичны из-за его модели программирования и реализации: потоки — это просто структуры данных, которые имеют особое значение для планировщика потоков. Межпотокое и межпамятное взаимодействие управляется явно, поэтому реализация указателей пересылки и обновлений указателей может быть выполнена с помощью эффективной и простой схемы, называемой «временно двунаправленные указатели». Наконец, использование системы памяти на основе возможностей с явной базой и границами тегированных областей памяти упрощает учет того, какие фрагменты памяти следует перемещать. Теперь становится разумным обсудить совершенно новый набор вопросов, связанных с онлайн-планированием миграции данных и потоков из-за этого механизма миграции с низкими накладными расходами.

## **4.2 Фон**

В этом разделе о фоновом режиме рассматриваются механизмы и алгоритмы предыдущей работы в области миграции данных и потоков. Этот раздел разделен на архитектуру, механизмы и алгоритмы разделы.

### **4.2.1 Архитектормеры, которые напрямую касаются миграции**

Существует несколько архитектур, которые напрямую решают проблему миграции данных или потоков. Класс архитектур, известный как СОМА (архитектура только кэш-памяти), должен напрямую бороться с проблемой миграции данных как проблемой размещения строк кэша. Машины NUMA (неоднородный доступ к памяти) также вводят идею пространственной осведомленности в архитектуру, но проблема миграции данных обычно инкапсулируется протоколом когерентности кэша. Механизмы миграции потоков, с другой стороны, обычно не проявляют себя как архитектурные особенности, а как поддерживаемые во время выполнения или компиляции особенности в остальном обычных параллельных архитектур. Поэтому в литературе механизмы миграции потоков обычно попадают в жанр механизмов перехвата работы и балансировки нагрузки и рассматриваются таким образом в следующем разделе.

В литературе относительно немного COMA. Наиболее известные COMA — это Bristol's Data Diffusion Machine (DDM) [MSW93], Kendall Square Research KSR-1 [ea92] и UIUC Illinois Aggressive COMA (I-ACOMA) [TP96]. Все три перечисленные здесь COMA полагаются на схему когерентности кэша на основе каталогов. KSR-1 и более поздние версии DDM используют масштабируемую иерархическую схему каталогов, тогда как опубликованная литература по I-ACOMA не указывает подробности схемы каталогов; на самом деле, литература по I-ACOMA не уделяет много внимания аспектам миграции данных COMA, а больше — схемам сокрытия задержек посредством использования одновременной многопоточности и ее реализации с использованием технологии обработки встроенной памяти. Как упоминалось ранее, COMA напрямую решают проблему миграции данных как проблему размещения строк кэша. В DDM кластер процессоров совместно использует «память притяжения» (AM), где хранятся запрашиваемые данные; часто запрашиваемые данные естественным образом мигрируют и кластеризуются вокруг процессоров, которым требуются данные. Местоположение данных отслеживается с помощью иерархического поиска в каталоге, основанного на двухточечном соединении, в отличие от KSR-1, который использует ряд взаимосвязанных колец для определения местоположения данных. Хотя иерархический поиск «точка-точка» решает некоторые проблемы масштабируемости взаимосвязанных колец KSR-1, он по-прежнему опирается на архитектуру поиска в каталоге. Это означает, что для хранения векторов битов присутствия в кэш-памяти необходимо либо большие строки кэша, либо высокие накладные расходы памяти. Хотя существуют такие механизмы, как разреженные каталоги [GWM90] или ограниченные указатели [ASHH88], которые могут уменьшить эти накладные расходы, эти механизмы вносят больше сложности в систему. С другой стороны, архитектура ADAM предоставляет программистам виртуальное общее пространство памяти и никаких кэшей. Согласованность в ADAM тривиальна, поскольку для любого изменяемого фрагмента памяти существует только одно местоположение; следовательно, никакая сложность или производительность не теряется в схеме кэширования каталога. Потеря производительности из-за отсутствия локального кэширования памяти компенсируется тремя способами. Первый — это простой сетевой протокол и архитектура, которые позволяют выполнять удаленные запросы памяти с низкой задержкой. Второй — агрессивная многопоточность для сокрытия задержек выборки в стиле HEP. [Smi82a] Третий — это использование механизмов миграции данных



и потоков, которые заменяют локальность данных, номинально предоставляемую схемами кэширования каталогов.

Архитектуры NUMA делают реальность неравномерного доступа к памяти явным архитектурным предположением и обычно предоставляют автоматические механизмы для сокрытия задержки удаленного доступа к памяти. В случае Stanford DASH [CDV 94] и SGI Origin 2000 протокол когерентности кэша на основе каталогов используется для улучшения локальности данных и повторного использования. Объем данных, которые могут быть «перенесены» локально в архитектуре ccNUMA, ограничен размером кэша. В отличие от DDM COMA, распределение, размещение и грубая миграция данных явно управляются в основном программным обеспечением; тем не менее, мелкозернистая миграция данных обеспечивается механизмом кэширования. Из-за больших накладных расходов, возникающих при управлении миграцией страниц программного обеспечения, эти машины ccNUMA попадают в класс машин крупнозернистой миграции данных. На этих машинах нецелесообразно рассматривать систему миграции, в которой данные динамически и часто перемещаются для уменьшения задержки и балансировки нагрузок. Например, SGI Origin 2000 обеспечивает аппаратную поддержку миграции страниц с помощью двух механизмов: счетчиков ссылок на каждую страницу для профилирования и механизма передачи блоков в стиле прямого доступа к памяти (DMA) для ускорения копирования страниц. Время, необходимое для копирования страницы памяти, составляет менее 30 с; однако время, необходимое для аннулирования и обновления TLB, составляет 100 с или более. [LL97] Хотя предоставляется метод, называемый «отравлением каталога», который позволяет обновлению TLB перекрываться с процессом копирования страниц, производительность копирования страниц все еще меньше, чем хотелось бы.

#### **4.2.2 Механизмы мягкой миграции**

Для эффективной миграции потоков с целью балансировки нагрузки в более традиционных архитектурах был предложен ряд инновационных высокопроизводительных механизмов.

TAM [CSS 91] (также именуемый Active Threads в [WGQH98]) и его продолжение, Active Messages [vCGS92], предлагают эффективный механизм для межпроцессорной связи с использованием продолжений. Он значительно отличается от J-Machine [NWD93], Monsoon

и \*T [PBB93], всех машин, управляемых сообщениями, тем, что Active Messages является чисто программным подходом к достижению высокой производительности. [vCGS92] утверждает, что чисто управляемые сообщениями аппаратные реализации ограничены ограниченным количеством регистров, доступных на аппаратный контекст, тогда как программно эмулируемая реализация может использовать богатую архитектуру обычного процессора. Он также отличается от других систем передачи сообщений тем, что работает полностью в пользовательском пространстве, чтобы исключить накладные расходы ядра, и допускает параллельную передачу сообщений и вычисления посредством неблокируемых операций. Active Messages продемонстрировал производительность 11 с (21 инструкция) для отправки сообщения и 15 с (34 инструкции) для получения сообщения на nCUBE/2. На CM-5 производительность составляет 1,6 с для отправки одного пакета (адрес + 16 байт сообщения) и 1,7 с для отправки получателю. Примечательно, что Active Messages не является механизмом миграции потоков; скорее, это метод для интеграции во время компиляции быстрых механизмов передачи сообщений, аналогичный по своей природе удаленным вызовам процедур (RPC). Таким образом, Active Messages не решает, как работать с пространственно неоднородной памятью или ситуациями, когда сложно статически проанализировать оптимальную схему создания потоков и обмена сообщениями.

Миграция вычислений — термин, введенный [HWW93]. Миграция вычислений похожа на миграцию потоков, но легче по весу (но не такая легкая, как потоки TAM). В этой статье подробно рассматривается разница между RPC, миграцией данных и миграцией вычислений. Для оценки жизнеспособности миграции вычислений использовалась прототипная система на основе PROTEUS (объектно-ориентированный язык) с явным программным аннотированием точек возможности миграции. Реализация была протестирована на счетной сети и эталонном тесте b-дерева. Производительность поддерживаемой оборудованием миграции вычислений благоприятна по сравнению с аппаратной общей памятью и поддерживаемой оборудованием RPC. Миграция вычислений особенно хороша в ситуациях с высокой конкуренцией. Возможно, наиболее интересным вкладом [HWW93] в отношении этой работы является подробная разбивка того, где тратится время в протоколе миграции. Из 651 цикла, необходимых для миграции вычислений, 74% потребляются «накладными расходами сообщений», т. е. перемещением памяти, планированием, маршалингом данных, созданием потоков и работой со связями процедур; только 3% потребляется при сетевом транзите, а оставшиеся 23% потребляются тем, что, по-

видимому, является аннотациями пользовательского кода. Аннотации пользовательского кода требуются в этой схеме, поскольку миграция явно управляется пользователем. Обратите внимание, что Active Threads [WGQH98], более медленная схема миграции, используется в качестве точки сравнения для моей работы над Computation Migration, поскольку эти статические аннотации ограничивают преимущества полезности и параллелизма Computation Migration. Несмотря на это, мой механизм миграции потоков работает примерно на порядок быстрее, цикл за циклом, чем схема Computation Migration. [Hsi95] описывает расширение работы, в котором динамическая миграция реализуется с использованием системы, называемой MCRL. Решения о миграции основаны на паре простых эвристик, основанных на частоте чтения и записи. Тесты, запущенные на системе MIT Alewife [ABC 95], показывают, что миграция вычислений может использоваться в сочетании с миграцией данных в ситуациях, когда записи в общую память являются обычным явлением для повышения производительности. ADAM расширяет эту работу, создавая аппаратный механизм для снижения накладных расходов на миграцию потоков и данных и, таким образом, позволяя эффективная мелкозернистая миграция.

Active Threads [WGQH98] — это документ, описывающий механизм миграции потоков, который использует схему потоков в пользовательском пространстве, схожую по духу с Cilk [Joe96], Filaments [LFA96] и Multipol [WCD 95]. Active Threads распределяет адреса узлов процессора по большому пространству виртуальной памяти, чтобы избежать необходимости обновлять указатели потоков при миграции потоков. Без специальной аппаратной поддержки Active Threads достигает односторонней задержки в 17 с для сообщения из 5 слов. Массовая передача 1 кбайт занимает 560 с, что ограничено пропускной способностью ввода-вывода хоста. Поток с нулевым стеком может быть перенесен за 150 с; на процессоре архитектуры Sparc v8 с использованием gcc 2.7.1 стек нулевого потока составляет 112 байт. Миграция стека размером 2 кбайт занимает 1,1 мс. Эти тесты проводились на кластере из 50 МГц Sparcstation 10 с Myrinet. В статье сравнивается этот механизм миграции потоков с такими схемами, как Ariadne, Millipede и PM; эти другие схемы имеют производительность порядка 10 мс для основных операций миграции. Наконец, демонстрируется сверхлинейное ускорение для миграции, управляемой локальностью, на простом многопоточном приложении `grep`, выполняющем поиск по распределенному дисковому массиву. Среднее время жизни потоков в этом тесте составляет

порядка 5-10 мс. Архитектура ADAM принимает использование Active Thread адресного пространства с чередованием узлов, но также повышает производительность, предоставляя аппаратный механизм для ускорения миграции и предоставляя временные двунаправленные указатели для выполнения ленивых обновлений указателей.

DEMOS/MP[PM83] — это операционная система, реализующая эффективный механизм миграции потоков. Механизм миграции потоков, описанный в DEMOS/MP, очень похож на тот, что используется в ADAM, но реализован полностью программно. Процессы DEMOS/MP состоят из состояния программы, таблиц ссылок, очередей сообщений и «другого состояния» (предположительно, состояния кучи). Межпроцессное взаимодействие происходит через выделенные ОС и администрируемые ссылки, которые записываются в таблицы ссылок. Такое использование явно управляемых межпроцессных связей обеспечивает эффективный механизм миграции процессов DEMOS/MP. Когда процесс хочет мигрировать, он останавливается, на удаленном узле выделяется место, и процесс перемещается. Сообщения, накопленные во время миграции, пересылаются в новое местоположение процесса, и существует механизм обновления таблиц ссылок отправителя для отражения нового местоположения процесса. В [PM83] мало упоминаний о производительности и нехватка сравнительных тестов, но в статье упоминается, что нулевой поток — поток без информации о программе или данных — имеет общий размер 850 байт. В статье также упоминается, что в нетривиальных процессах размер областей данных и информации программы намного больше размера нулевого потока. Таким образом, можно смело предположить, что накладные расходы на миграцию в DEMOS/MP довольно высоки, поскольку ее процессы примерно эквивалентны по структуре процессам, обнаруженным в современных системах UNIX. Архитектура ADAM улучшает механизм миграции DEMOS/MP, используя облегченное представление потока, которое перемещается быстрее, и предоставляя архитектуру, которая обеспечивает аппаратную поддержку механизмов межпроцессного взаимодействия. Миграция потоков в ADAM также не требует перемещения состояния кучи или обхода таблиц распределения памяти на основе ОС. Архитектура и механизм миграции ADAM также позволяют осуществлять миграцию данных в дополнение к миграции потоков.

### 4.2.3 Среда программирования и алгоритмы онлайн-миграции

Проектирование аппаратного механизма будет неполным без продуманной среды программирования или алгоритмов, необходимых для использования возможностей механизма.

Emerald [JLHB88] — основополагающая работа в области систем миграции объектов. Единственными другими работами, цитируемыми в этой работе, являются распределенная реализация Smalltalk, Argus и Eden; можно также считать Hydra и Clouds (объектно-ориентированные операционные системы) предыдущими работами. Emerald — это системный проект, воплощающий язык и реализацию. В языке есть система типов, которая позволяет программисту давать подсказки компилятору. Он также обеспечивает подсказки по миграции, распределению и средству в языке. Emerald также является сборщиком мусора. Язык использует глобальное уникальное пространство имен.

Объекты могут иметь прикрепленные к ним процессы или они могут быть прямыми данными; решение о прикреплении процесса к объекту принимается компилятором. Emerald уделяет большое внимание поддержанию хорошей производительности локального вызова, несмотря на предоставление возможности переносить объекты. Указатели пересылки с временными метками используются в качестве метода для быстрой миграции объектов без необходимости перетаскивать вселенную вместе с перемещаемым объектом. Решение о том, какие части объекта перемещать, принимает среда выполнения и компилятор; небольшие фрагменты данных перемещаются во время миграции; более крупные фрагменты требуют большего обдумывания. Emerald также предоставляет глобальную возможность поиска объектов. Одной из проблем Emerald является обработка регистров процессора: несогласованность может привести к состоянию регистра процессора из-за способа перемещения записей активации. В статье было продемонстрировано, что Emerald имеет хорошую производительность по сравнению с немигрирующей реализацией распределенного приложения для обработки почты. Наконец, в статье дается хорошее резюме преимуществ миграции: распределение нагрузки, производительность связи, доступность, реконфигурация и простота использования специальных возможностей.

Ciurpe, Kottman и Walter [CKW96] предлагают фреймворк для включения управляемой программистом миграции объектов в своей статье под названием «Миграция объектов в монолитных распределенных приложениях». В статье утверждается, что объектно-

ориентированная модель является естественным соответствием для миграционной структуры, поскольку объекты естественным образом определяют локальность данных и методы, которые могут ее изменять. В статье предполагается, что основными лингвистическими примитивами, необходимыми для управления миграцией, являются операции фиксации, операции перемещения и нотации прикрепления. В статье также предполагается, что все высокоуровневые решения о миграции кодируются «разумными пользователями». Языковые примитивы тестируются в абстрактной среде моделирования, которая делает такие предположения, как полностью связанная сеть. Моделирование показывает, что немая миграция (базовая кодируемая пользователем миграция) дает примерно такое же увеличение производительности, как и миграция на основе профиля. Результаты также показывают, что миграция может быть пагубной в ситуациях, когда политики миграции кодируются только с учетом одного компонента. В частности, производительность снижается в случае горячих точек и в случае, когда рабочий набор объектов тесно связан, но мигрирует как отдельные сущности.

«Profiling Based Task Migration» Бакстера и Пателя [BP92] фокусируется только на миграции для балансировки нагрузки и имеет очень специфичный, ограниченный набор данных. Однако он демонстрирует, что для этого конкретного примера алгоритм миграции, действующий только на локальных знаниях, может достичь производительности в пределах 5% от решения на основе глобальных знаний.

Kalogeraki, Melliar-Smith и Moser [KMSM01] обсуждают динамические алгоритмы для миграции распределенных объектов в своей статье под названием «Алгоритмы динамической миграции для систем распределенных объектов». В этой работе рассматриваются системы всего с 8 узлами и примерно 5 объектами на узел. Перенос состояния объекта затруднен перемещением состояния ОС/ядра в рамках архитектуры распределенных объектов ORB [Inc01]; планирование объектов происходит за миллисекунды, профилирование — за секунды, а миграция — за десятки секунд. Тестовая система — ULTRA Sparc 167 МГц с использованием VisiBroker ORB 3.3, а межсоединение — Ethernet 100 Мбит/с. Основное внимание в статье уделяется использованию миграции для удовлетворения ограничений системы реального времени, поэтому результаты показывают, что «слабость» может быть сохранена посредством миграции. Таким образом, соответствующий раздел этой статьи для этой диссертации — ее динамические алгоритмы миграции. В статье представлены алгоритмы «охлаждения» (балансировки нагрузки) и

«горячей точки» (сокращения задержки), оцененные независимо. Алгоритмы соответствуют интуиции, которую несут их названия, и в статье показано, что эти алгоритмы могут быть использованы для успешной балансировки задачи в распределенной системе.

Система Object Request Broker (ORB) [Inc01], используемая [KMSM01], описана в документе объемом более 1000 страниц. ORB — это открытая архитектура и спецификация для определения объектов, которые могут совместно использоваться, взаимодействовать и вызываться под одним огромным общим зонтиком. Как отмечалось ранее, накладные расходы, возникающие в системе ORB, помещают ее в другую лигу систем миграции по сравнению с этой диссертацией; однако сам стандарт решает ряд интересных вопросов программирования, которые выходят за рамки этой диссертации.

В контексте распределенных объектных систем реального времени [HS94b] использует байесовский анализ и теорию очередей для определения того, следует ли переносить задачу на узел назначения, учитывая набор ограничений реального времени и некоторые оценки времени выполнения и неопределенности задачи. Статья ссылается на предыдущую работу [HS94a], в которой описывается, как оценить состояние загрузки удаленного узла с учетом устаревшей информации. В этой статье основное внимание уделяется обеспечению того, чтобы решения о переносе работы на другой узел принимались таким образом, чтобы учитывались и будущие поступления задач. Это предотвращает ситуацию, когда все отправляют свои задачи на один незагруженный узел в сети только потому, что устаревшая информация о загрузке выглядела хорошо на момент начала миграции. В этой статье также представлена идея «приятелей», которые физически расположены совместно для ограничения диапазона широковещательной информации о состоянии, и попытки сократить объем коммуникаций, необходимых для поддержания состояния.

Эта статья является хорошим примером формального анализа миграции данных в сложной системе с использованием статистического анализа. Другие методы анализа системы могут быть через теорию систем управления (системы с обратной связью) или через конкурентный анализ в режиме реального времени. Существенное отличие этой статьи от работы, которой я занимаюсь, заключается в том, что эта работа исследует системы реального времени, тогда как моя работа просто интересуется оптимальной производительностью (минимальное время выполнения в отличие от гарантированного времени выполнения).

Холл и др. [ННК 01] представляют теоретическую статью о миграции данных в контексте балансировки нагрузки и оптимизации системы хранения. Предполагается полностью связанная двунаправленная сеть с объектами одинакового размера. Даже при этих предположениях задача определения оптимального плана миграции данных объявляется NP-полной. Задача также является NP-полной для всего двух узлов, напрямую связанных с объектами переменного размера, учитывая, что только один объект может перемещаться в любой момент времени, и что пространство на каждом узле очень ограничено. В статье утверждается, что эта задача эквивалентна раскраске ребер для задачи неограниченного пространства и очень похожа по решениям на задачу раскраски ребер для задач ограниченного пространства. Хорошей новостью является то, что доступны эвристики и поливременные алгоритмы, которые могут решить задачу почти до оптимальности. [BEY98] — это обзорная работа по конкурентному анализу и онлайн-алгоритмам, в которой описываются некоторые алгоритмы, которые можно применять к задачам миграции данных и балансировки нагрузки. Значительную часть формального анализа в моей диссертации я основываю на содержании [BEY98].

### **4.3 Реализация механизма миграции**

Q-Machine — это реализация абстрактной архитектуры ADAM. Q-Machine использует архитектурные особенности ADAM для обеспечения быстрых механизмов миграции с низкими издержками. Этот механизм снижает задержку и стоимость полосы пропускания при миграции легких данных и потоков до уровня заполнения кэша L2 на процессоре Pentium 4 в системе на базе RAMBUS. Расчетная системная задержка заполнения кэша L2 составляет около 175 нс (что составляет 140 циклов Direct-RAMBUS 800 МГц) [CJDM01], а размер строки кэша L2 составляет 128 байт [HSU 01]. Обратите внимание, что кэш L2 процессора Pentium 4 разделен на 64-байтовые половины, но согласно [HSU 01], заполнение кэша L2 «обычно» извлекает данные для обоих секторов. Более подробную информацию о производительности механизма миграции можно найти в разделе 6.

Вспомогательные детали реализации Q-Machine представлены в главе 5; сейчас я сосредоточусь исключительно на деталях реализации, относящихся к миграции данных и потоков. Также при чтении этого раздела предполагается, что читатель знаком со спецификацией архитектуры ADAM (глава 3 и приложение В).



Сердцем механизма миграции является архитектура возможностей с тегами ADAM и использование карт очередей для межпроцессного взаимодействия. Эти две дисциплины, обеспечиваемые аппаратным обеспечением, радикально сокращают объем бухгалтерского учета и специального оборудования, необходимого для реализации эффективного механизма миграции. Возможности кодируют свою базовую и связанную информацию в своих тегах, поэтому границы перенесенных данных являются явными. Кроме того, возможности в этой архитектуре имеют бит «только приращение», который позволяет безопасно резервировать части начала возможности для служебных функций, таких как пересылка указателей и ведение статистики. Кроме того, состояние потока имеет однозначное отображение с возможностью (контекстным идентификатором потока) в памяти из-за реализации файла очереди именованного состояния (подробности реализации файла очереди см. в приложении С). Эта функция позволяет миграции потока совместно использовать почти все механизмы миграции данных; основное отличие заключается в том, что миграция потока требует дополнительной блокировки и синхронизации с физическим файлом очереди. Использование карт очередей для межпроцессного взаимодействия важно, поскольку оно обеспечивает простые механизмы синхронизации, перенаправления и обновления запросов межпотокowego взаимодействия во время и после события миграции потока.

#### **4.3.1 Механизм удаленного доступа к памяти**

Теперь я представлю механизм удаленного доступа к памяти, используемый в реализации Q-Machine. Механизм удаленного доступа к памяти является важным компонентом механизма миграции. Напомним, что адресное пространство ADAM структурировано таким образом, что идентификатор узла процессора является наивысшими битами адреса; также, по соглашению, узлы процессора занимают четные адреса маршрута, а узлы памяти занимают нечетные адреса маршрута. Это позволяет объединять процессоры и память в «предпочтительные» пары посредством существования надежного, упорядоченного сетевого пути доставки между предпочтительными парами. Таким образом, локальный доступ к памяти определяется как доступ к памяти, при котором идентификатор узла возможности доступа равен идентификатору узла предпочтительного узла памяти. Локальный доступ к памяти всегда обслуживается предпочтительным узлом памяти, а запросы на выделение локальной памяти выделяют данные в предпочтительном узле

памяти. Производительность доступа к данным в предпочтительном узле памяти аналогична производительности времени доступа к кэшу L3 на современном процессоре; конкретные цифры см. в разделе 6.

Семантически предпочтительный узел памяти является целью всех отображений очередей MML, MMS и EXCH, независимо от возможности доступа, используемой для инициализации отображения. Таким образом, все удаленные запросы также направляются с узла процессора на предпочтительный узел памяти. Когда инициализируется удаленный запрос, локальный обработчик виртуальной памяти выделяет локальные «теньевые» страницы для удаленной возможности. Теньевые страницы выполняют две функции: во-первых, они предоставляют метод для хранения указателя локатора данных удаленной памяти; во-вторых, они предоставляют инфраструктуру для кэширования неизменяемых данных. Теньевые страницы никогда не должны замещать локальные страницы памяти, когда локальной памяти не хватает. Следовательно, большинство тневых страниц не выгружаются в ядро или не инициализируются при первом выделении. Единственным исключением является первая страница. Первое расположение памяти первой тневой страницы — это указатель локатора данных. Этот указатель локатора данных инициализируется с возможностью удаленного доступа. Обратите внимание, что остальная часть пространства первой тневой страницы помечена как недействительная и неосновная. На рисунке 4-1 показан формат удаленной возможности в тневом пространстве.

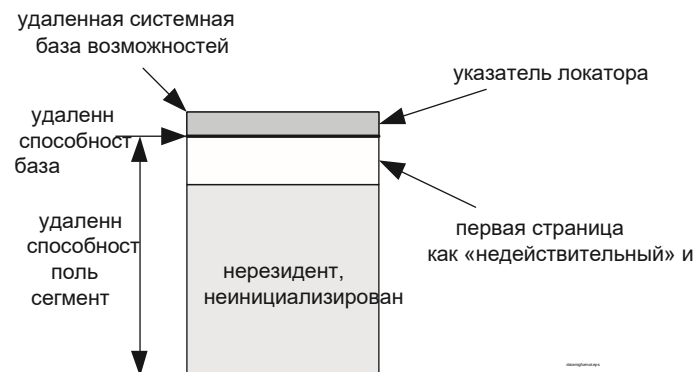


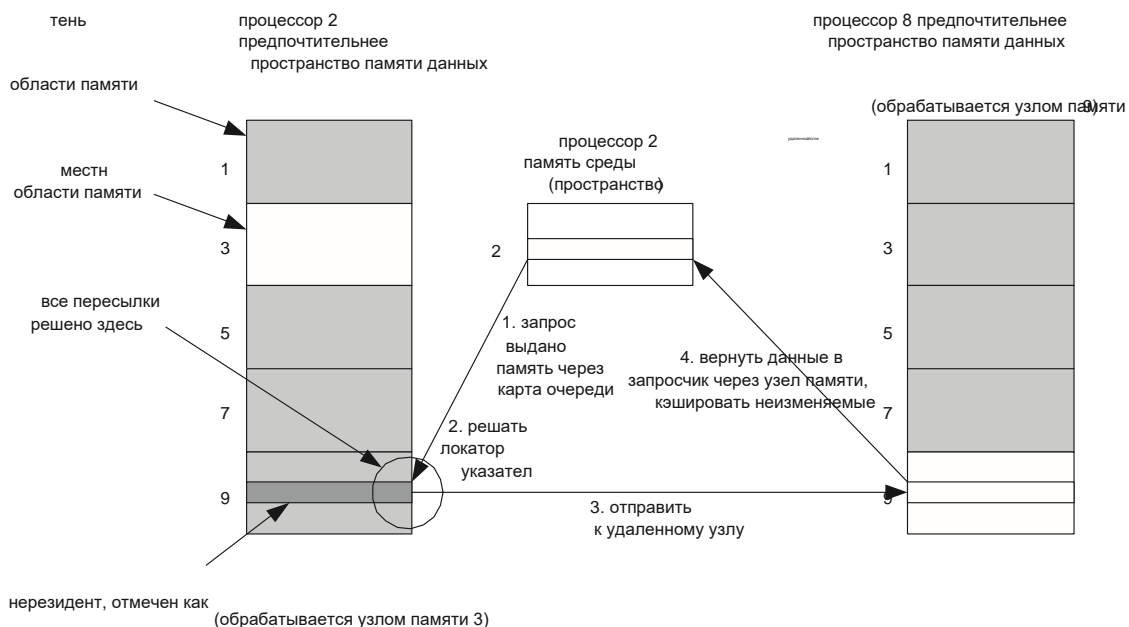
Рисунок 4-1: Формат теневого пространства удаленной памяти в локальном виртуальном пространстве памяти.

Рисунок 4-2 представляет обзор системного уровня разрешения удаленного запроса памяти. Удаленные запросы легко обнаруживаются, когда очередь, отображенная в память,

инициализируется с ее возможностью доступа: если идентификатор узла возможности доступа не равен идентификатору узла памяти, это должен быть удаленный запрос. Этот статус удаленного запроса отмечается в тегах таблицы доступа узла памяти (для получения дополнительной информации о таблице доступа узла памяти см. раздел 5.3.2).

Все запросы от узла процессора к предпочтительному узлу памяти используют формат транспортного пакета без заголовка маршрута физического уровня и контрольных сумм. Более подробную информацию о транспортном протоколе, используемом в Q-Machine, можно найти в приложении С.2. Эти транспортные пакеты содержат все состояния, необходимые для разрешения обратного адреса запрашивающей стороны; таким образом, при пересылке запроса памяти узел памяти просто инкапсулирует исходный пакет запроса процессора в заголовки пересылки и отправляет инкапсулированный пакет на удаленный узел памяти. Пожалуйста, см. рисунок 4-3 для более подробной иллюстрации того, как обрабатываются сопоставления локального и удаленного обмена (EXCH). Операция EXCH была выбрана для иллюстративных целей, поскольку она объединяет как операцию загрузки, так и операцию сохранения. Операция загрузки использует ровно половину загрузки протокола EXCH, а операция сохранения использует половину сохранения протокола EXCH, а также пакет подтверждения сохранения, чтобы можно было гарантировать завершение записи в программном порядке.

Обратите внимание, что для совместимости с механизмом миграции, который будет описан в следующем разделе,



все местоположения  
неосновные на странице

Рисунок 4-2: Вид на системном уровне разрешения удаленных запросов памяти.

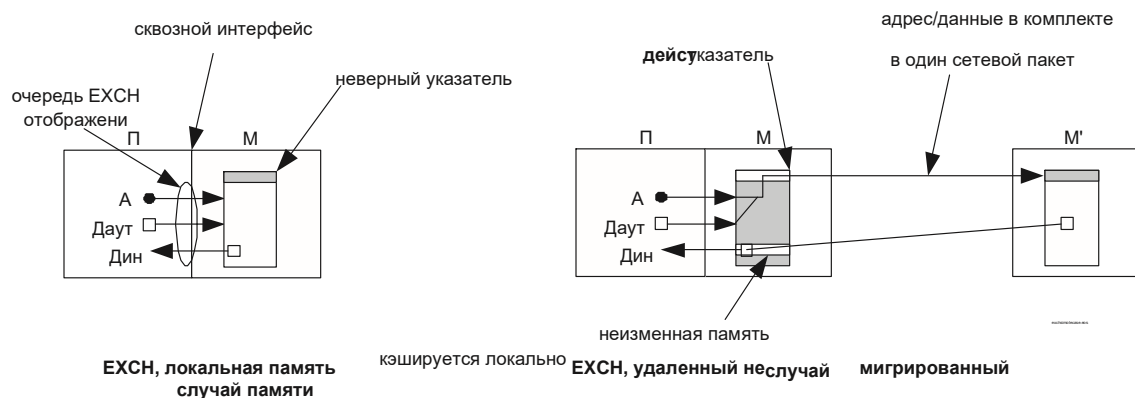


Рисунок 4-3: Подробная информация об обработке удаленных и локальных запросов EXCH.

все доступы к памяти, даже хранилища, должны проверять допустимые и основные биты тегов. Если существующее значение недействительно и установлен неосновной бит (как в случае миграции данных), то таблица доступа должна быть обновлена для пересылки будущих запросов, а текущий запрос также должен быть переслан в удаленную очередь запросов. Накладные расходы на проверки тегов для всех запросов, включая хранилища, можно уменьшить, если в реализации памяти предусмотрено выделенное оборудование.

### 4.3.2 Механизм миграции

Примитивы удаленного доступа к памяти, описанные в предыдущем разделе, позволяют оптимизировать реализацию механизмов миграции. Механизм миграции распознает только две команды,

миграция данных ( ) и перенести поток ( ). Аргументы к этим

Команды — это исходная способность данных или потока для миграции и идентификатор процессора назначения. Другие команды, которые могут быть реализованы, включают команды частичной миграции и команды копирования неизменяемых данных.

### 4.3.3 Миграция данных

Механизм миграции данных, реализованный в Q-Machine, основан на следующих предположениях и инвариантах.

**Инвариант:***Пользователь видит только одно глобальное уникальное имя для каждой возможности, и это имя никогда не меняется.* Это обеспечивается указателем базового локатора данных в верхней части каждой возможности. Этот локатор данных позволяет фактическим данным свободно перемещаться без необходимости одновременного изменения состояния потока.

**Предположение:***В любой момент времени на каждый узел памяти может приходиться максимум один исходящий процесс миграции.* Это предположение упрощает требования к оборудованию для заморозки и синхронизации запросов на доступ к передаваемым данным.

**Инвариант:***Относительный порядок запросов к любой заданной возможности сохраняется до, во время и после миграции.* Это важно для поддержания согласованности модели памяти.

**Предположение:***Относительный порядок запросов между мигрирующими и немигрирующими возможностями не важен.* Это общее предположение архитектуры, но оно здесь переформулировано для ясности. Например, запрашивающая сторона несет ответственность за то, чтобы сохранение в одном месте было завершено до загрузки в то же место. В реализации Q-Machine допускается только один ожидающий запрос на поток в уникальном месте памяти; более производительное решение может использовать буферы хранения с ассоциативным поиском для устранения этого узкого места, пока это не вызовет проблем со следующим предположением.

**Предположение:***В любой момент времени на каждый поток в каждой уникальной ячейке памяти в сети может быть только один ожидающий запрос памяти.* Это довольно ограничительное предположение необходимо, поскольку запросы к возможности миграции задерживаются на время события миграции; фактически, в случае, если данные мигрируют через узкое место маршрутизации, запросы, отправленные после миграции, придут раньше любых ожидающих запросов, отправленных до миграции. Можно ослабить это

предположение с помощью дополнительного учета в механизме миграции и протоколе обновления указателя, но этот вид сложности оптимизации производительности избегается в моем исследовательском прототипе. Обратите внимание, что локальные запросы имеют менее ограничительные требования, поскольку интерфейс cut-through имеет более сильные гарантии упорядочения запросов, чем внешний сетевой интерфейс.

**Инвариант:***В любой момент времени в системе существует максимум одна основная копия возможности.* Первичная копия возможности — это копия, которой разрешено отвечать на запросы загрузки изменяемых данных или любые запросы на сохранение или обмен. Возможность является первичной, когда биты первичного тега установлены для всех данных в сегменте возможности.

**Инвариант:***Если в системе нет первичных копий возможности, запросы к этой возможности не обслуживаются.* Другими словами, передаваемые данные не могут быть изменены или прочитаны.

**Предположение:***Возможность миграции начинается локально.* Узел памяти не может управлять миграцией возможностей, которые не являются локальными; если это необходимо, локальный узел памяти должен отправить сообщение удаленному узлу памяти для запроса миграции.

**Совет по повышению производительности:***Полезно иметь аппаратный механизм для очистки или установки первичных битов в больших блоках памяти.* Это частая операция, выполняемая механизмом миграции, которая плохо масштабируется с размером сегмента возможностей. Одним из подходов к реализации может быть чередование операции очистки первичного бита со считыванием данных во время фазы копирования миграции.

Это процедура миграции возможности.

Запрос на миграцию оформляется в формате

Запрос на выделение возможности      подходящего размера выдается получателю памяти узел.

Запросы на обслуживание до тех пор, пока возвращается в исходный узел.

Все входящие сетевые запросы замораживаются с использованием механизма, схематически представленного на рисунке 44. Обратите внимание, что исходящие

данные могут продолжать отправляться и повторно отправляться с помощью идемпотентного последовательного транспортного протокола, описанного в разделе С.2.

копируется в

Содержание отмечены как недействительные и непервичные, за исключением неизменяемых данных.

Запись локатора данных изменено с недействительного на .

Исходящие данные в полете до заморозки должны быть все подтверждены в соответствии с протоколом последовательной идемпотентной сети перед переходом к следующему шагу. Запросы на размораживаются и перепланируются. Эти запросы теперь обрабатываются существующей инфраструктурой удаленного доступа к памяти. В конце концов, после обновления всех указателей, механизм сборки мусора освобождает память.

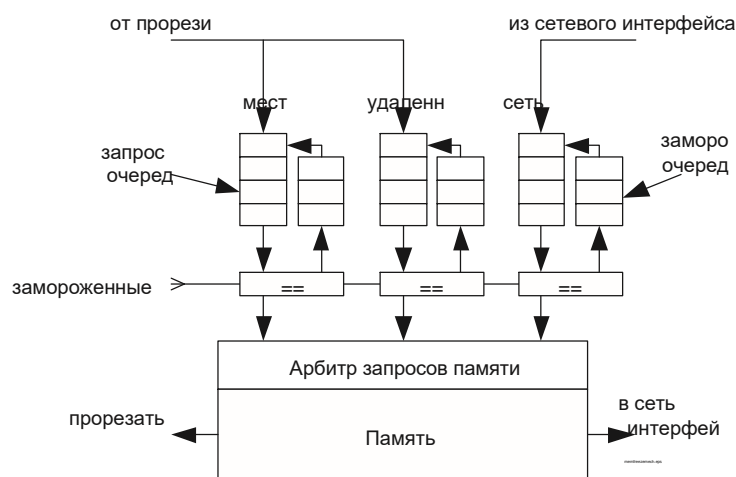


Рисунок 4-4: Механизм временной заморозки запросов памяти.

В дополнение к механизму миграции требуется механизм для обновления входящих указателей локатора данных, иначе каждый запрос памяти в конечном итоге должен будет пройти по цепочке указателей локатора данных. Один из методов выполнения обновлений указателей — это прочесть память и разрешить все указатели локатора данных в их основных местоположениях. Этот метод непомерно дорогой и медленный. Лучшим решением является использование двунаправленных указателей локатора данных и отправка сообщений об обновлении по обратным путям каждый раз, когда часть данных мигрирует. Однако двунаправленные указатели имеют недостаток, заключающийся в необходимости поддерживать произвольно большой список обратных указателей.

Обновление обратного указателя также забивает исходящие сетевые порты узла памяти, если список обратных указателей большой.

Чтобы противостоять этим ошибкам, я использую механизм, который называю временными двунаправленными указателями.

Временно двунаправленные указатели можно рассматривать как лениво вычисляемые двунаправленные указатели.

Всякий раз, когда отправляется запрос на перенесенную возможность, ответом является сообщение об обновлении указателя.

Это сообщение об обновлении указателя содержит тело исходного запроса, так что запрашивающему не нужно отслеживать состояние невыполненного запроса. Пожалуйста, обратитесь к рисунку 4-5. Хотя механизм временных двунаправленных указателей тратит впустую одну поездку по сети на обновление по сравнению с активно обновляемыми двунаправленными указателями, временные двунаправленные указатели имеют много преимуществ: они требуют постоянного пространства для реализации; обратные указатели, которые неактивны, не потребляют ресурсов; запросы на обновление распределяются во времени, поэтому эффективная задержка ниже из-за меньшей очереди запросов; и, если блок данных был перемещен через узкое место, узкое место не усугубляется потоком сообщений об обновлении.

#### 4.3.4 Миграция потока

Механизм миграции потоков, реализованный в Q-Machine, основан на следующих предположениях и инвариантах.

**Инвариант:** *Пользователь видит только одно глобальное уникальное имя для каждого потока, и это имя никогда не меняется.* Это обеспечивается указателем локатора данных в верхней части каждой возможности, включая возможности потока. Этот локатор данных позволяет фактическим данным свободно перемещаться без необходимости одновременного изменения состояния потока.

**Инвариант:** *Все межпоточные операции выполняются только для записи.* Это бесплатно с моделью ADAM «push» межпоточных коммуникаций. Другими словами, разрешены только исходящие карты очередей; локальная очередь не может запрашивать «чтение» данных из удаленной очереди.



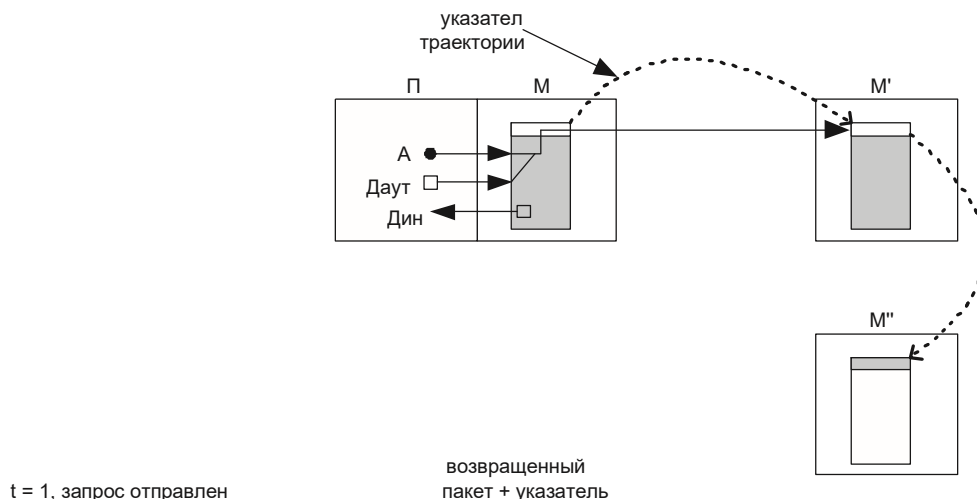
**Предположение:** В любой момент времени на каждый процессорный узел приходится максимум один исходящий процесс миграции. Это предположение упрощает требования к оборудованию для заморозки и синхронизации запросов к потоку в процессе выполнения.

**Инвариант:** Относительный порядок запросов к любому потоку сохраняется до, во время и после миграции. Это важно для поддержания согласованного порядка данных в сопоставленных очередях.

**Предположение:** Относительный порядок запросов между мигрирующими и немигрирующими потоками не важен. Это общее предположение об архитектуре, но для ясности оно здесь перефразировано.

**Инвариант:** В любой момент времени в системе существует максимум одна основная копия потока. Первичная копия потока — это копия, которая может быть запланирована и является допустимой целью для входящих данных из сопоставленных очередей. Поток является первичным, когда первичный бит установлен в идентификаторе контекста. Напомним, что идентификатор контекста также является возможностью для резервного хранилища потока.

**Инвариант:** Если в системе нет активных копий потока, запросы к потоку не обслуживаются. Другими словами, поток, находящийся в процессе выполнения, не может быть изменен или прочитан.



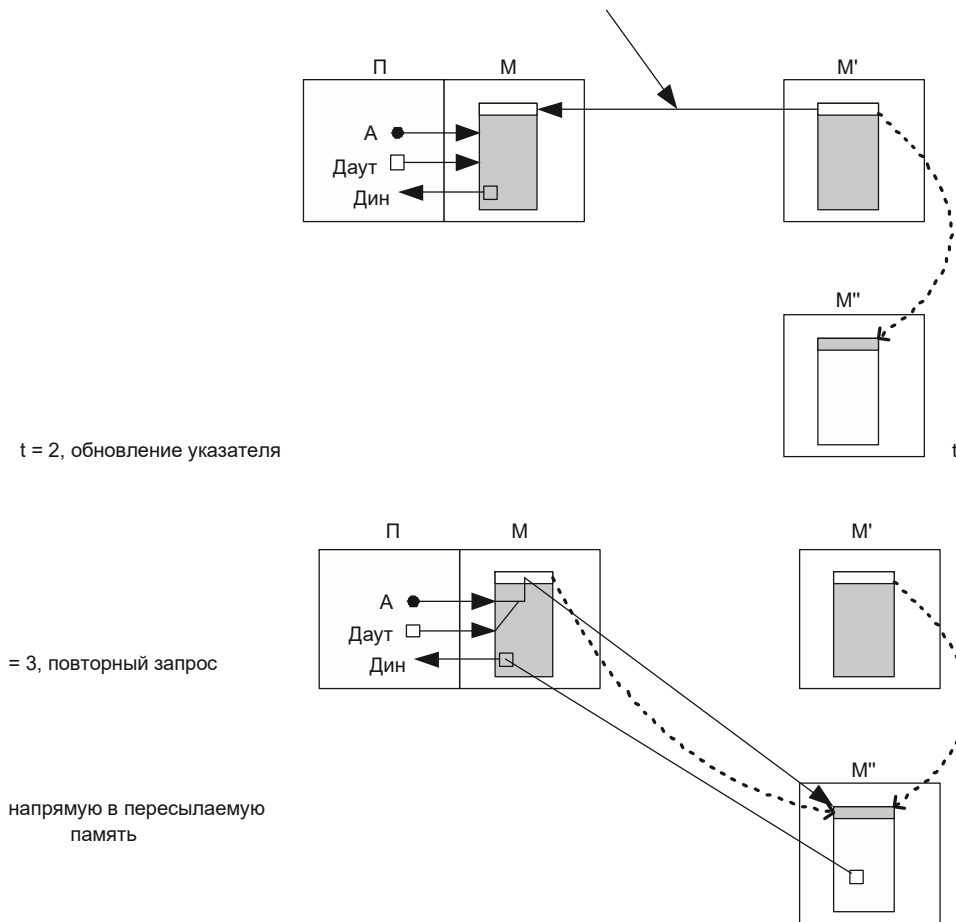


Рисунок 4-5: Обработка перенесенного запроса EXCH с временными двунаправленными указателями.

**Предположение:** Поток, который необходимо перенести, начинается локально. Узел процессора не может управлять миграцией потоков, которые не являются локальными; если это необходимо, локальный узел процессора должен отправить сообщение с запросом на миграцию удаленному узлу процессора.

**Совет по повышению производительности:** Аппаратное обеспечение должно отслеживать созданные очереди, а также то, к каким именно очередям применены карты памяти, карты источников и карты отбрасывания. С этой информацией очереди, которые не были созданы (т. е. никогда не упоминались или иным образом были пустыми), могут потреблять нулевые накладные расходы во время миграции. Реализация файла очереди

именованных состояний Q-Machine предоставляет всю эту бухгалтерскую информацию бесплатно.

Q-Machine реализует две процедуры для миграции потоков. Одна используется, когда поток определяется как «легкий», т. е. у него мало отображений памяти, мало созданных очередей и мало других состояний, связанных с ним. Другая используется, когда поток определяется как имеющий большой объем состояний и может вызвать проблемы с загрузкой в сети и приемнике; это называется «тяжелым» потоком. Основное различие между протоколами заключается в синхронизации удаленного выделения потока по сравнению с поступлением состояния потока. Тяжеловесная миграция потока выдает запрос на выделение перед отправкой данных потока; это позволяет подготовке к миграции происходить параллельно с обслуживанием запроса на выделение. Легковесная миграция потока объединяет состояние потока с удаленным запросом на выделение потока; эта оптимизация сокращает время, необходимое для миграции легковесного потока.

Ниже приведена процедура миграции облегченного потока; ответственность за обработку этой миграции делится между отправителем и получателем.

#### **Процедура отправителя для облегченной миграции потока:**

Запрос на миграцию оформляется в формате

Поток удаляется из локального пула потоков. Это включает в себя очередь работы процессорного узла и любое внутреннее состояние, поддерживаемое планировщиком потоков.

Все входящие запросы на                      заморожены высоздавайте механизм, аналогичный показанному на рисунке 4-4.

Все из                      Состояние сбрасывается из файла физической очереди в память среды. Все ожидающие запросы памяти для                      необходимо завершить или, по крайней мере, подтвердить перед выполнением следующего шага.

Все ожидающие исходящие запросы на транспортном уровне должны быть подтверждены перед переходом к следующему шагу.

Состояние потока переносится в узел                      .  
                    установлен как неосновной.

После получения сообщения «миграция прошла успешно» от , ожидающие запросы на могут быть разморожены и перепланированы. Механизм обновления указателя, обсуждаемый далее, обрабатывает эти запросы.

В конце концов, после всех указаний на были обновлены, механизм сбора мусора освобождает .

### **Процедура получателя для облегченной миграции потока:**

Получен входящий пакет миграции, содержащий состояние потока.

выделяется, и Состояние потока копируется в .

Очереди, отображенные в памяти, реконструируются для предпочтительного узла памяти получателя.

немедленно помещается в пул исполняемых потоков получателя.

Отправителю отправляется токен «миграция прошла успешно».

Ниже приведена процедура миграции тяжелого потока. Опять же, бремя протокола делится между отправителем и получателем.

### **Процедура отправителя для миграции тяжеловесного потока:**

Запрос на миграцию оформляется в формате

Запрос на выделение возможности подходящего размера выдается принимающему процессору

узел (узел ).

Поток удаляется из локального пула потоков. Это включает в себя очередь работы процессорного узла и любое внутреннее состояние, поддерживаемое планировщиком потоков.

Все входящие запросы на замораживаются с помощью механизма, аналогичного показанному на рисунке 4-4.

Все из Состояние сбрасывается из файла физической очереди в память среды. Все ожидающие запросы памяти для необходимо завершить или, по крайней мере, подтвердить перед выполнением следующего шага.

Все ожидающие исходящие запросы на транспортном уровне должны быть подтверждены, прежде чем перейти к следующему шагу.

необходимо получить, прежде чем переходить к следующему шагу.

состояние потока - мигрированный узел .

установлен как неосновной.

После получения токена «миграция успешна» от узла ожидающие запросы могут быть разморожены и перепланированы. Механизм обновления указателя, обсуждаемый далее, обрабатывает эти запросы.

В конце концов, после всех указаний на были обновлены, механизм сбора мусора освобождает .

### **Процедура получателя для облегченной миграции потока:**

Получен входящий запрос на выделение потока; пространство выделяется и возвращается к отправитель.

Состояние потока принимается и копируется в .

Очереди, отображенные в памяти, реконструируются для предпочтительного узла памяти получателя.

помещается в пул работающих потоков.

Отправителю отправляется токен «миграция прошла успешно».

Обновления указателя отображения очереди обрабатываются немного иначе, чем обновления указателя локатора данных для миграции данных, поскольку во время миграции потока допускается наличие нескольких запросов, ожидающих выполнения в одной очереди. В этом случае используется протокол «линии передачи». Название было выбрано, чтобы отразить сходство этой ситуации с распространением и отражением волн в линии передачи с последовательным подключением. См. рисунок 4-6. Этот протокол опирается на одно дополнительное предположение.

**Предположение:***Все сообщения между отправляющим и получающим потоками гарантированно доставляются и обрабатываются в порядке, соответствующем последовательности сообщений, сгенерированной отправляющим потоком.* Это предположение поддерживается идемпотентным последовательным транспортным протоколом, используемым в реализации Q-Machine.

Это протокол линии передачи:

Поток заморожен из-за выполняющейся миграции

Входящие запросы к замороженному потоку блокируются, а пакет «обновления указателя пересылки», содержащий заблокированные входящие запросы, возвращается отправителю, как только поток будет перенесен.

После того, как отправитель получает пакет «обновление указателя пересылки», он отправляет сообщение «подтверждение пересылки» и прекращает отправку дальнейших запросов к перенесенному потоку; в то же время отправитель повторно отправляет все возвращенные запросы в новое местоположение потока.

Как только старое местоположение получает пакет «подтверждение пересылки», оно отправляет отправителю пакет «разрешено разблокировать».

Получив пакет «разрешено разблокировать», отправитель может отправлять новые запросы в перенесенный поток.

## **4.4 Проблемы и наблюдения механизма миграции**

Подробный обзор эффективности механизма миграции можно найти в разделе 6. В этом разделе отражены некоторые из моих общих наблюдений о механизме миграции и проблемах его проектирования и реализации.

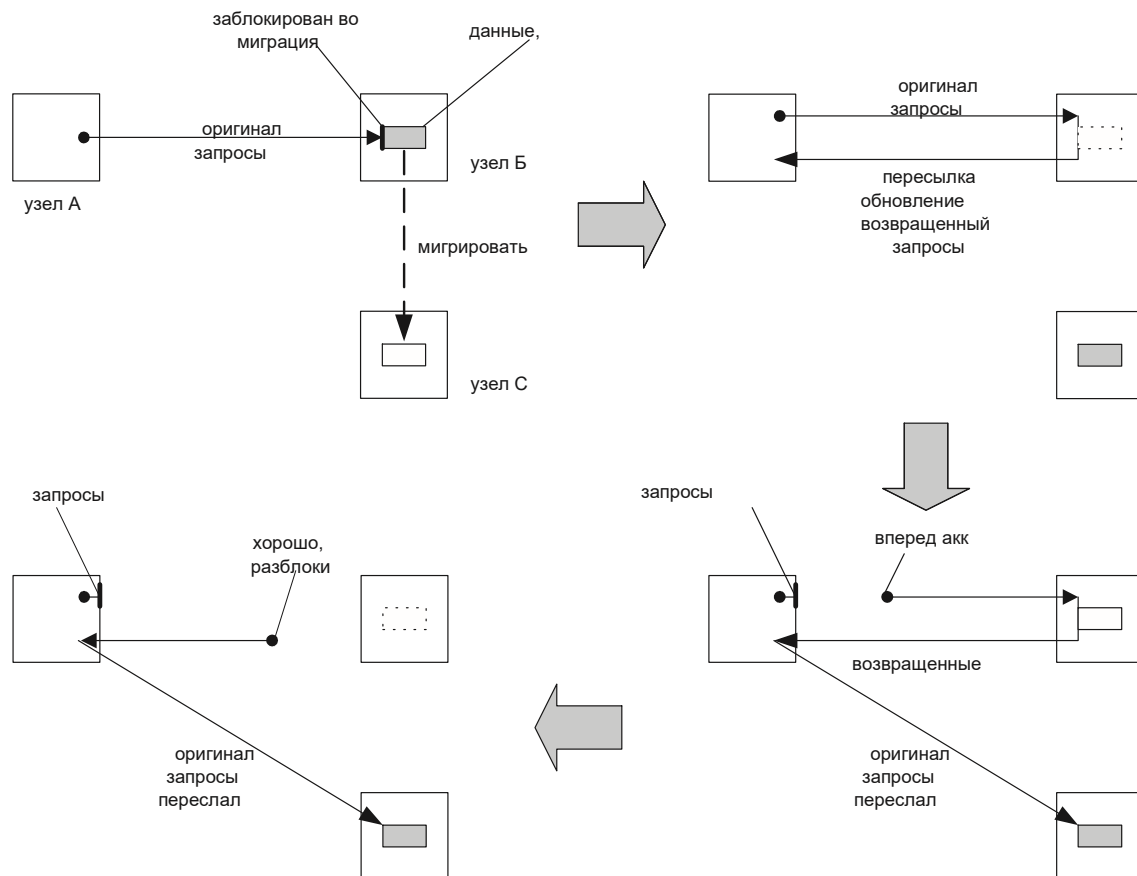
### **4.4.1 Общие замечания**

Базовый протокол миграции в архитектуре на основе возможностей с собственной поддержкой указателей пересылки прост: заблокировать возможность, переместить данные, затем разблокировать возможность. Конечно, дьявол кроется в деталях. Именно сложные детали приводят к дихотомии возможностей памяти и потоков. Другими словами, хотя и указывалось, что, возможно, один механизм может быть использован как для потоков, так и для данных, похоже, что характер того, как память и потоки используются программистами, приводит к естественному разделению этих структур в архитектуре машины.

Одним из основных компромиссов, которые можно получить, обрабатывая данные и потоки отдельно, является упрощение протокола миграции памяти. Поскольку память не имеет вспомогательного состояния — нет файла очереди, нет записей планировщика, только один ожидающий запрос на поток в каждом месте — перемещение памяти требует гораздо меньших накладных расходов, чем перемещение потока. С другой стороны, поскольку потоки используют строго однонаправленную связь, обновления указателей можно

управлять несколькими невыполненными операциями. Несколько одновременных операций с памятью сложнее, поскольку необходимо поддерживать порядок между всеми возможными очередями хранения, загрузки и обмена, которые могут быть динамически сопоставлены с одним местом памяти. Таким образом, разделив машину на отдельные узлы памяти и процессора, соответствующие протоколы миграции могут отсеять любые ненужные предположения или условия, специфичные для каждой ситуации.

Кроме того, модель программирования только с потоками может быть выгодна в ситуациях, когда производительность зависит от наличия нескольких одновременных запросов к памяти. В модели программирования только с потоками программист не видит узлов памяти или очередей, отображенных в памяти; вместо этого выделенные потоки сервера обрабатывают запросы памяти абстрактным образом. ADAM может быть спе-



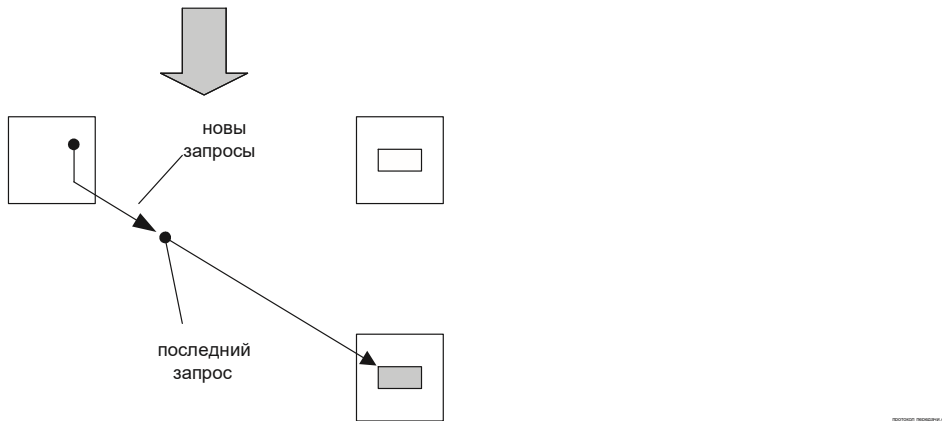


Рисунок 4-6: Протокол линии передачи для обработки обновлений указателей пересылки в потоковых сообщениях.

сialized в модель программирования только для потоков с использованием трюков компилятора с некоторой поддержкой ОС. Компилятор заботится о вставке кода, необходимого для создания абстрактных выделенных серверов памяти, а ОС отвечает за координацию миграции потоков сервера вместе с их связанными данными.

#### 4.4.2 Проблемы с производительностью

Я разработал механизм миграции как высокопроизводительное решение с задержками и требованиями к пропускной способности, сопоставимыми с заполнением кэша L2 в современном обычном процессоре. Результаты, представленные в разделе 6, показывают, что я достигаю этой цели для легких потоков и данных. Конечно, есть еще оптимизации, которые можно применить к механизму миграции.

Например, частичная миграция состояния потока может ускорить время между решением о миграции и первым прибытием потока в пункт назначения. Моя схема блокирует поток и перемещает все его содержимое перед повторным планированием потока. Более сложная схема будет отслеживать последний и наиболее часто используемый набор данных в потоке и просто отправлять эти данные в небольшом пакете для немедленного планирования; поскольку принимающий узел инициализирует и планирует этот поток для выполнения, память среды может быть одновременно заполнена оставшимися данными потока. Частичная миграция осуществима только из-за гибкости в реализации файла очереди именованных состояний, используемой Q-Machine.

Частичная схема миграции также может быть применена к большим наборам данных. Если выделяется чрезвычайно большая возможность, возможность может быть разделена на



подблоки, представляя фактическую возможность как набор меньших возможностей внутри большой возможности. Меньшие подблоки будут содержать указатели на родительскую возможность, и наоборот; реализация указателя будет похожа на схему указателя локатора данных, используемую моим механизмом удаленного доступа к памяти. Эти подблоки будут свободно перемещаться по системе независимо от родительской возможности.

Другим методом частичной миграции является копирование данных по требованию. В этой схеме в дополнение к указателям локатора данных также требуется набор обратных указателей. «Основная» возможность по-прежнему отвечает за обработку всех операций с возможностью, но данные для загрузки или обмена распространяются на основную возможность только по запросу. Эта схема проиллюстрирована на рисунке 4-7. Этот метод может быть полезен, если очень разреженные, большие структуры данных часто мигрируют по всей машине.

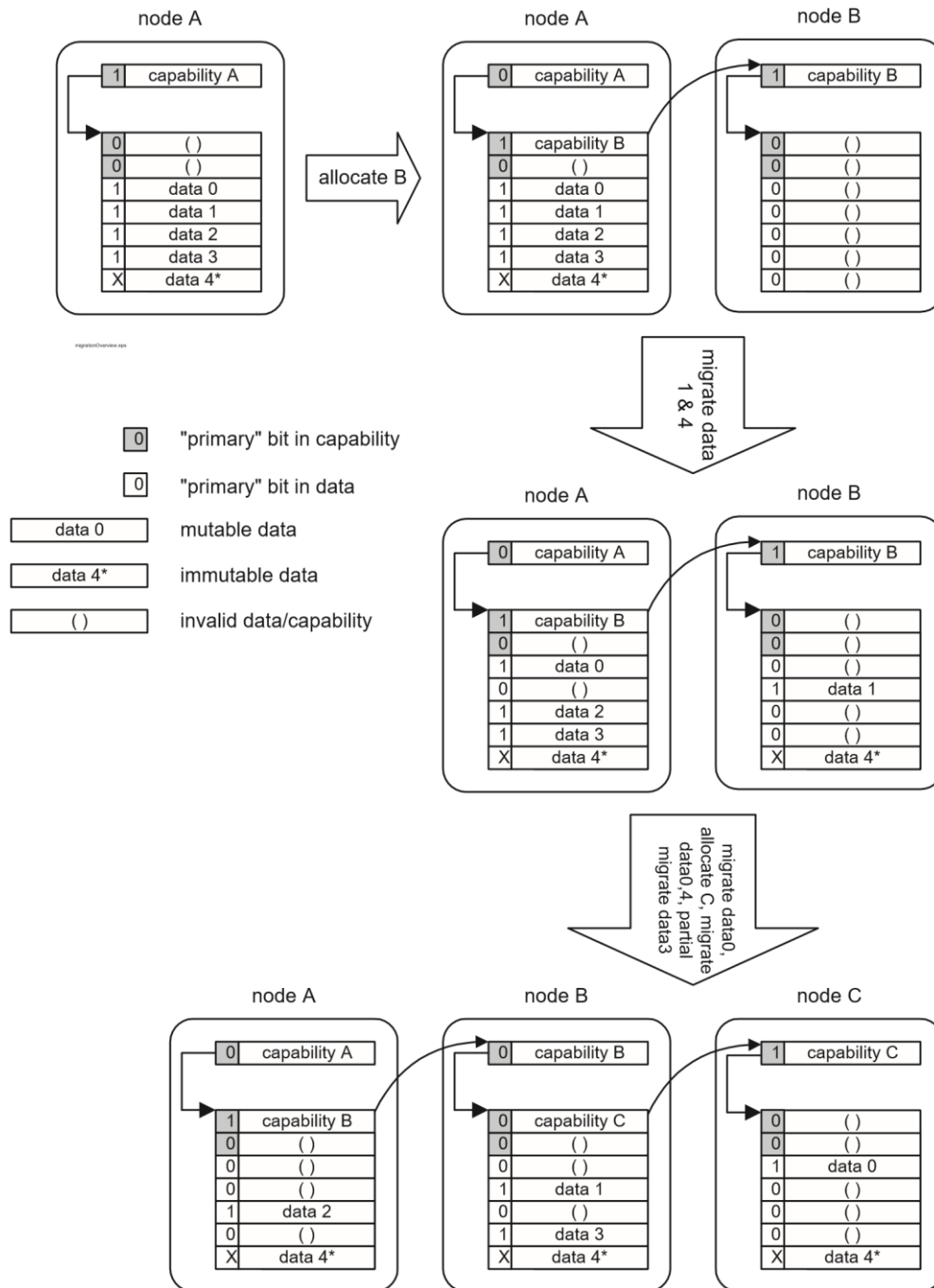
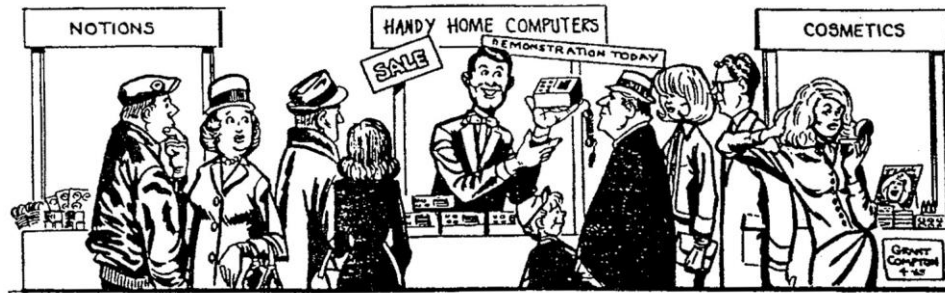


Рисунок 4-7: Обзор схемы распространения данных по запросу.

## Глава 5

# Реализация ADAM:

# Аппаратное обеспечение и моделирование



*Из книги Гордона Мура «Втиснуть большие компоненты в интегральные схемы»,  
апрель 1965 г.*

В этом разделе описываются детали реализации физической машины, называемой Q-Machine, оптимизированной для запуска кода ADAM. При описании реализации используется подход сверху вниз. Все ключевые архитектурные особенности проверяются кратким исследованием осуществимости, чтобы сохранить соответствие проекта реальности и попытаться убедить читателя в том, что это реализуемая архитектура. В этом разделе также описывается, как предлагаемые аппаратные параметры отражаются в разработке и реализации программного симулятора Q-Machine. Программное моделирование является точным по пропускной способности и задержке и почти точным по циклам. Первая цель моделирования — продемонстрировать осуществимость модели программирования на основе очередей, используемой ADAM, и продемонстрировать интеграцию с языками и инструментальными цепочками Couatl и People. Вторая цель моделирования — продемонстрировать производительность механизмов миграции потоков и данных, описанных в разделе 4, и предоставить платформу для тестирования различных алгоритмов миграции.

## 5.1 Введение

Q-Machine организован как мелкозернистый параллельный процессорный массив плиток MIMD со встроенной памятью. Преобладание предлагаемых архитектур плиточных процессоров или чип-мультипроцессоров (CMP), многие из которых имеют встроенную

RAM той или иной формы, недавно возникло из-за их привлекательной простоты и соблазнительных обещаний производительности «гарантировано не превышать». Некоторые из этих недавно предложенных архитектур включают RAW [LBF 98], Hydra [HNS 00], IRAM [KPP 97], MAJC от Sun Microsystem, IBM Power4, Active Pages [OCS98], Decoupled Access DRAM [VG98], Terasys [GHI94], SPACERAM [Mar00], Smart Memories [MPJ 00] и Hamal [Gro01]. Краткая статья Кунле Олукотуна суммирует основные преимущества CMP. [ONH 96]

## 5.2 Организация высокого уровня

Адресное пространство Q-Machine разделено на три части: код, среда и данные. Пространство кода предназначено для однократной записи, многократного чтения, а данные распределяются по всем узлам; взаимодействие между пространством кода и пространством пользователя возможно только через код операции LDCODE. Пространства среды и данных предназначены для многократного чтения, многократной записи, а их адресные пространства являются локальными для каждого узла. Пространство среды — это место, где хранятся контексты потоков; таким образом, все взаимодействие с пространством среды является неявным. Среда «выделяется» набором кодов операций SPAWN, а потоки «сборщика мусора», поскольку они ОСТАНАВЛИВАЮТСЯ или наблюдаются как больше не ссылающиеся или не ссылающиеся на что-либо еще в системе. Доступ к пространству данных осуществляется только через сопоставления очередей в исполнительном блоке. Для обработки запросов памяти в пространстве памяти требуется сопроцессор управления памятью. Код операции ALLOCATE предоставляется в наборе инструкций как средство для создания памяти, а для освобождения памяти требуется механизм сборки мусора. Взаимодействие кода операции ALLOCATE с сопроцессором управления памятью зависит от реализации. Рисунок 5-1 иллюстрирует ситуацию высокого уровня, которая приводит к такому разделению адресных пространств. Обратите внимание на специфические для реализации параметры, такие как устройства ввода-вывода и пользовательские аппаратные блоки на рисунке 5-1. К этим устройствам можно получить доступ либо через сопоставления

очереди, установленные ловушками ОС, либо через коды операций, добавленные к спецификациям стандартного ADAM, которые ведут себя аналогично инструкциям ALLOCATE и SPAWN.

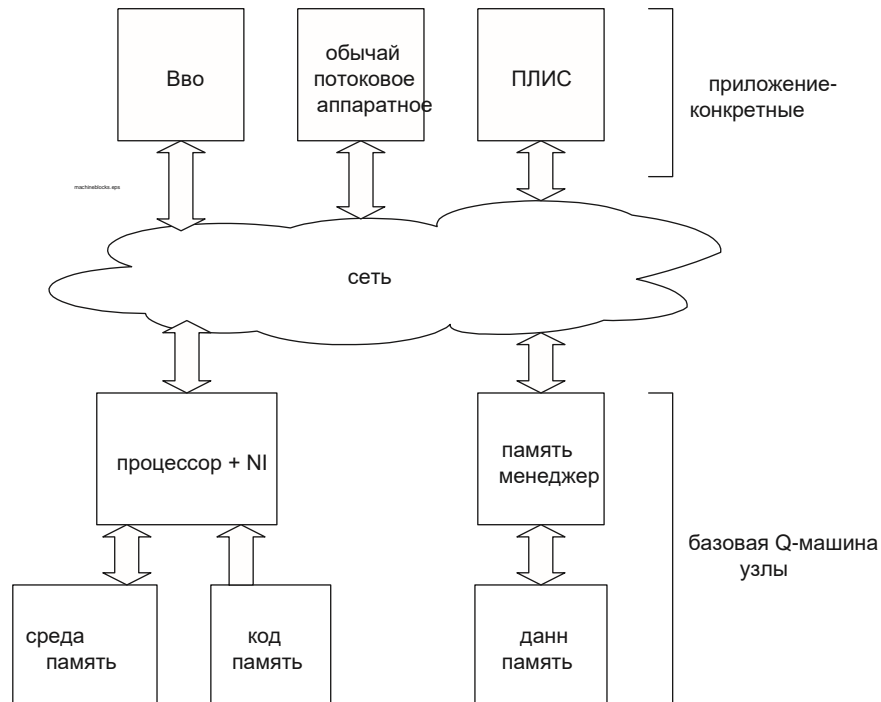


Рисунок 5-1: Фрагменты реализации Q-машины. Теги идентификаторов узлов одинаковы по всей машине, поэтому подключенное к сети пользовательское оборудование адресуемо, как любой процессор или узел памяти.

### 5.3 Листовой узел

Базовый конечный узел содержит два основных узла: узел процессора и узел памяти. Каждый из этих узлов выглядит идентичным первичной сети с точки зрения маршрутизации и адресации. Однако между каждой парой узлов процессор-память предусмотрен сквозной путь с низкой задержкой. Этот путь устанавливает связанный узел памяти как предпочтительное местоположение данных, с которыми узел процессора хочет работать. Сквозной путь гарантированно всегда надежно доставляет данные между парой конечных узлов; таким образом, можно избежать задержки, связанной с добавлением заголовков пакетов, контрольных сумм блоков и других операций учета, возникающих в транспортном протоколе надежной доставки. Должны быть достаточные пропускная способность и порты, чтобы сделать вероятность перегрузки процессора или узла памяти сквозным трафиком

пренебрежимо малой. Простой способ гарантировать это предположение — разбить конструкцию так, чтобы был выделенный порт для сквозного трафика, отдельный от портов для обработки межузлового трафика. Разбиение таким образом сопряжено с риском выделения избыточных ресурсов на недостаточно используемый сквозной порт; Однако задача менеджера миграции и планировщика заключается в том, чтобы попытаться структурировать распределение данных и вычислений таким образом, чтобы максимально использовать локальность. Блок-схему реализации единичного листового узла можно найти на рисунке 5-2.

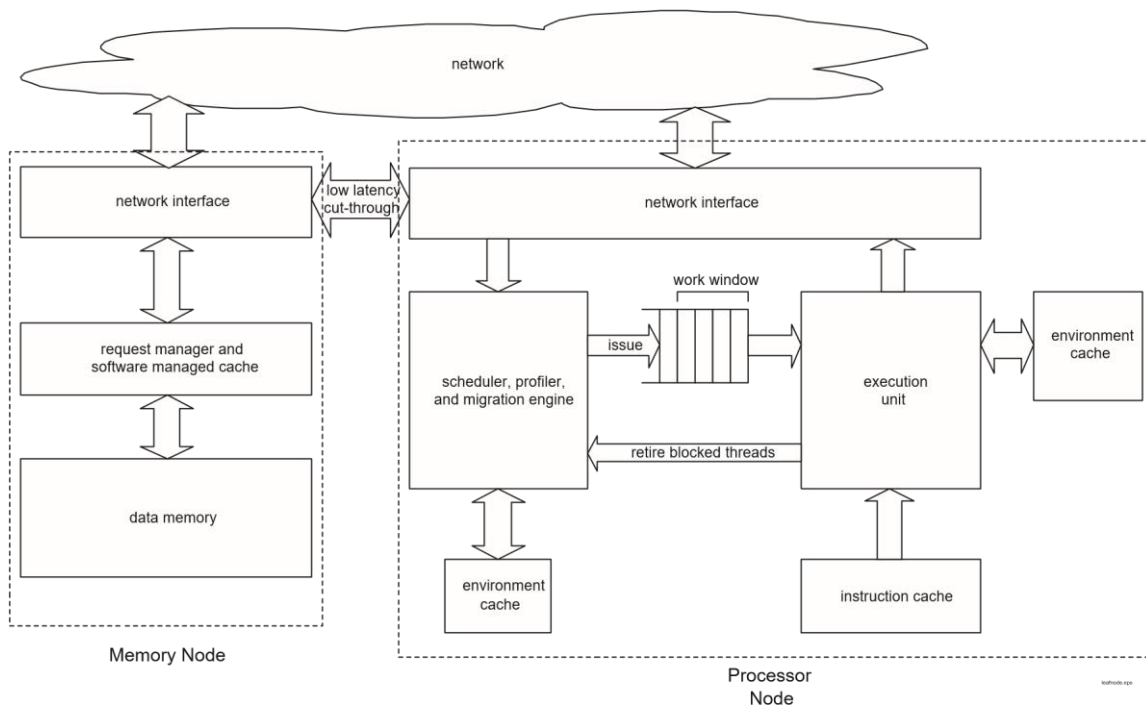


Рисунок 5-2: Высокоуровневая структурная схема листового узла.

### 5.3.1 Узел процессора

Процессорный узел состоит из пяти основных подкомпонентов: исполнительного блока, планировщика, сетевого интерфейса, кэша среды и кэша инструкций. Обзор организации процессорного узла можно найти на рисунке 5-3.

Планировщик и исполнительный блок взаимодействуют через путь рабочего окна и путь отставленного потока. Рабочее окно представляет собой небольшой буфер запланированных потоков для запуска. Каждый запланированный поток связан со строкой кэша инструкций,

которая предварительно извлекается, пока поток ожидает в рабочем окне. Симулятор системы ADAM реализует рабочее окно глубиной в восемь потоков. Исполнительный блок поддерживает указатель, который вращается через рабочее окно всякий раз, когда поток блокируется. Исполнительный блок также поддерживает счетчик длины запуска каждого элемента в рабочем окне и заставляет элемент выгружаться при достижении максимальной длины запуска, чтобы предотвратить голодание других потоков. Максимальное количество запусков программируется во время выполнения.

Когда поток блокируется, он может быть удален из рабочего окна и отправлен обратно в планировщик с тегом, указывающим, на каком фрагменте данных поток заблокировался. Метод определения того, когда заблокированный поток должен быть удален, не является жестко запрограммированным.

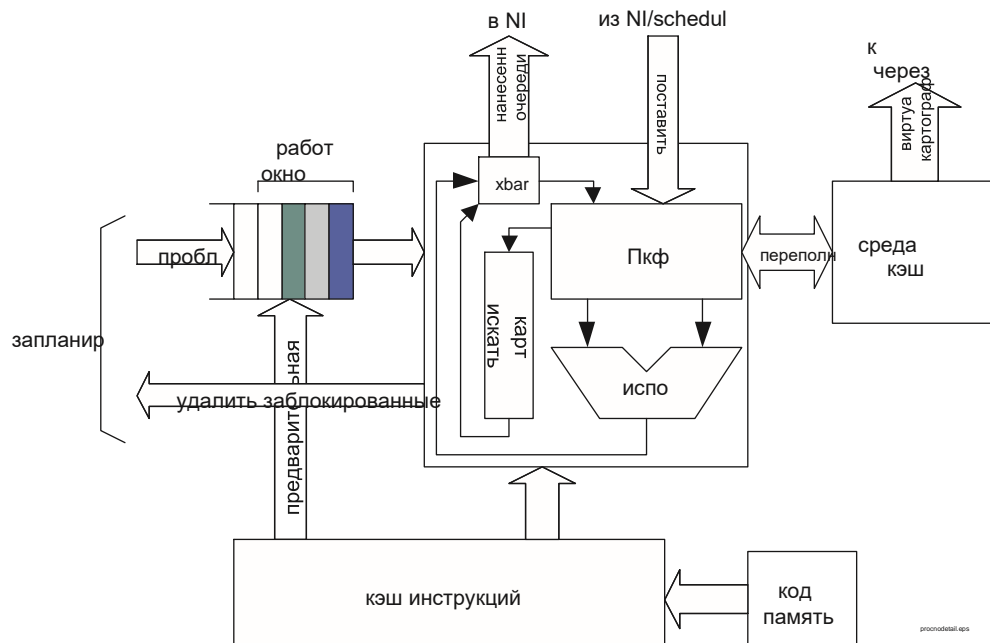


Рисунок 5-3: Деталь процессорного узла.

## Ядро процессора

Ядро процессора само по себе похоже на классическую архитектуру RISC. Операнды извлекаются из файла физической очереди (PQF) и отправляются непосредственно в арифметическое устройство (EXEC); результаты записываются обратно, по большей части, в PQF. PQF реализован способом, аналогичным файлу именованного регистра состояния (NSRF) [ND95]. Таким образом, PQF можно рассматривать как кэш для состояния потока.

PQF полностью ассоциативен: любая строка в PQF может быть сопоставлена с любой очередью в любом контексте потока. Непрерывная область пространства памяти данных выделена для «памяти среды», так что строки в PQF имеют простое взаимно-однозначное сопоставление с адресами в памяти. Эта память среды гарантированно является локальной для узла через контракт с менеджером миграции (см. Раздел 4 для получения более подробной информации). Одной из отличительных особенностей PQF является то, что он имеет функцию автоматического сброса, так что когда PQF превышает определенный порог заполнения, строки удаляются только при наличии доступной полосы пропускания для кэша среды. ADAM System Simulator реализует PQF со 128 строками и порогом автоматического сброса в 124 строки. Пожалуйста, см. Приложение С для получения дополнительных заметок о реализации PQF.

Межпотокное взаимодействие происходит только через модель «push». Модель push означает, что поток может генерировать только межпоточный трафик, нацеленный на другой узел; он не может «вытягивать» данные из другого узла, размещая межпотокное сопоставление на порте чтения. Заставляя потокное взаимодействие происходить только при записи данных, поиск сопоставления очереди может происходить параллельно с вычислением результата. Таким образом, критические накладные расходы пути сводятся к минимуму для межпоточного взаимодействия. Данные, предназначенные для контекста потока, расположенного на локальном процессорном узле, немедленно возвращаются в локальный планировщик, который выполняет некоторую бухгалтерию, а затем быстро пересылает данные непосредственно в PQF.

Важное наблюдение заключается в том, что когда рабочий набор контекстов имеет след, который вписывается в PQF, и мало межпоточной коммуникации, путь данных ядра выполнения выглядит почти так же, как у стандартного процессора RISC. Эта простота критического пути позволяет разработчику легче достигать высоких тактовых частот в ядре выполнения и, в свою очередь, обеспечивать высокую производительность однопоточного кода. Также обратите внимание, что ядро выполнения можно легко расширить до суперскалярной реализации проблемы вне порядка: структура очереди файла регистров дает некоторое количество переименования регистров бесплатно, а пустые биты в PQF упрощают реализацию внеочередной отправки.

## **Планировщик**



Использование мелкозернистой многопоточности для сокрытия задержки ранее наблюдалось в Tera/MTA [AKK 95], HEP [Smi82a], M-Machine [FKD 95] и \*T [PBB93], среди прочих. Алгоритм планирования, реализованный в симуляторе для этой работы, является производным от алгоритма, использованного в [NWD93], и следует общему алгоритму планирования, описанному во введении к этому разделу. Потоки делятся на два пула: пул работоспособных и пул остановленных. Пул работоспособных выполняется по принципу циклического перебора с тайм-аутом упреждения потока для гарантии некоторой справедливости. Потоки, которые блокируются при остановке доступности данных, удаляются в пул остановленных; потоки, которые блокируются при структурной остановке (например, при промахе файла очереди именованного состояния), перемещаются в нижнюю часть пула работоспособных. Поток перемещается из остановленного пула в готовый к работе пул, когда данные потока поступают через сетевой интерфейс. Все входящие данные должны проходить через сетевой интерфейс, поскольку единственный механизм доставки данных потоку — это сопоставления очередей, а все сопоставления очередей маршрутизируются через сетевой интерфейс.

В Q-Machine предусмотрены специальный планировщик и сопроцессор профилирования, чтобы исключить накладные расходы на выяснение того, какие потоки запускать и когда переносить объекты из ядра выполнения. Планировщик работает только с локально доступной информацией и использует небольшой банк локальной памяти, поэтому его реализация намного легче, чем ядро выполнения. Другими словами, планировщику не требуются механизмы межпоточной связи на основе очередей, реализованные в ядре выполнения, поэтому он может использовать более простой файл регистров и механизмы прямого доступа к памяти загрузки/хранения. Таким образом, планировщик реализован как слегка усовершенствованный 16- или 32-разрядный RISC-процессор, который работает полностью из нескольких мегабит локальной памяти. В реализации, выпущенной в 2010 году (подробнее об этом в разделе 5.4), предполагается, что память объемом 5 мегабит очень легко реализуется в быстрой технологии SRAM; если используется технология DRAM, емкость может быть в 10 или 20 раз больше. Программируемый сопроцессор планировщика выбирается вместо выделенного оборудования, поскольку проблема планирования и миграции очень сложна и ее трудно реализовать непосредственно на оборудовании. Кроме того, амбициозный пользователь или компилятор может захотеть настроить код

планировщика, если ожидаются очень регулярные или предсказуемые шаблоны работы потоков.

Основное аппаратное усовершенствование сопроцессора планировщика — это прямой интерфейс к пустым/полным битам потоков, ожидающих планирования; входящие данные из межпоточного трафика должны пройти через планировщик, прежде чем быть записанными в PQF (или в кэш среды, если PQF заполнен), так что планировщик знает, какие потоки стали разблокированными в результате поступления ожидающих данных. Другое аппаратное усовершенствование планировщика — это быстрый прямой интерфейс к списку планировщика. Список планировщика тесно связан с кэшем инструкций. Каждая запись списка планировщика содержит строку I-кэша, указатель на строку, который отражает точное значение счетчика программы, идентификатор контекста и поле указателя, которое, если оно действительно, содержит указатель на следующую запись списка планировщика. Неиспользуемые записи списка планировщика можно рассматривать как общие строки I-кэша и наоборот. Пример гибридной структуры планировщик/I-кэш показан на рисунке 5-4. Чтобы поддерживать высокую скорость I-кэша, можно использовать меньший кэш-буфер, выделенный только для I-кэширования, или, возможно, разработчик может разделить поля тега и идентификатора контекста для обеих функций и использовать менее ассоциативное, но более быстрое сравнение для строк, помеченных как помеченные. Планировщик выполняет только поиск на основе индекса для элементов планировщика, поэтому ему не требуется ассоциативный компаратор, а требуется более длинное поле тега, поскольку несколько потоков часто будут проходить через один и тот же фрагмент кода. Обратите внимание, что функции I-кэша и планировщика можно сделать взаимоисключающими, разделяя одно и то же физическое пространство без слишком большого влияния на критический путь индексации и поиска I-кэша: строки сравнения кэша могут иметь вывод «match» компаратора, закрытый битом режима «sched»; таким образом, строки, выделенные для функциональности планировщика, выглядят для функции кэширования как недействительные строки.

Ожидается, что кэш инструкций будет иметь емкость в несколько тысяч строк, поэтому для того, чтобы узел вошел в блокировку планировщика, должно быть запущено около нескольких тысяч потоков. Если емкость структуры планировщика считается слишком маленькой для размещения в оборудовании, дополнительный бит может

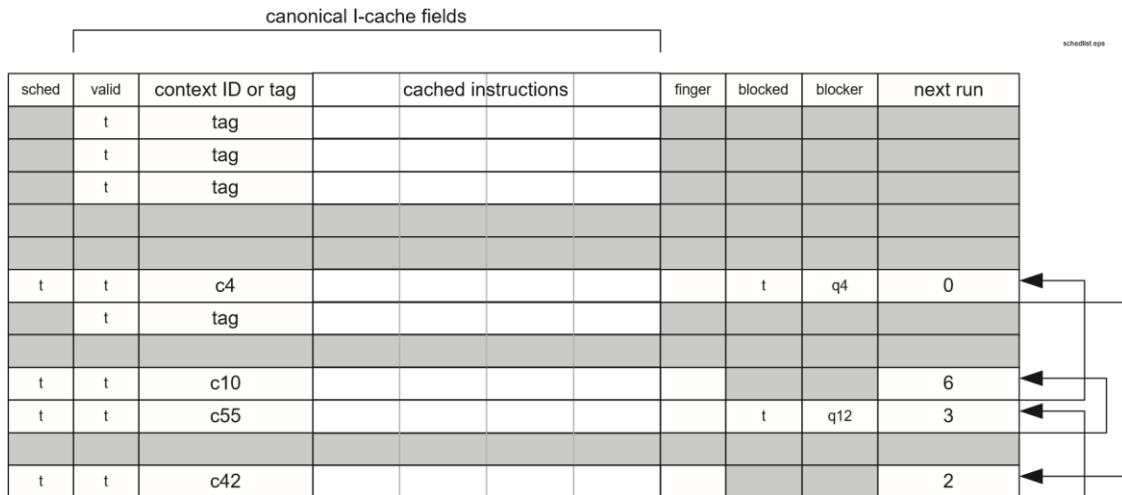


Рисунок 5-4: Гибридная структура списка планировщика/I-кэша. На этой диаграмме c42 и c10 готовы к запуску и пересылке в очередь работ; по мере поступления значений для c55:q12 и c4:q4 через NI они будут переведены в статус готовых к запуску.

быть предоставлено в поле «следующая исполняемая строка», что заставляет сопроцессор планировщика принимать ловушку при запросе следующей исполняемой строки и вместо этого проверять явно управляемую основную память на основе связанный список.

Гибридная структура планировщика/списка предпочитает иметь по крайней мере два порта записи и два порта чтения, по одному для функции планировщика и для функции I-cache; однако реализация может обойтись одним портом чтения/записи, если короткие простои терпимы. Хотя на первый взгляд трафик планировщика может показаться очень малым по сравнению с трафиком строки кэша, сопроцессор планировщика также отвечает за изменение порядка связанного списка готовых к запуску элементов в зависимости от приоритета и статуса элемента. Он также отвечает за вставку и удаление запланированных элементов по мере того, как потоки порождаются, собираются мусором или переносятся в узел и из него.

## Сетевой интерфейс

Обсуждение сетевого интерфейса, используемого в Q-Machine, отложено до приложения С.

### 5.3.2 Узел памяти

Узел памяти реализован как сетевой интерфейс для большого банка DRAM плюс небольшой сопроцессор, который помогает координировать параллельные и атомарные операции. Он также может помочь управлять локальным кэшем данных, чтобы улучшить время доступа и увеличить среднее количество запросов, на которые был дан ответ за сетевой цикл. Сетевой интерфейс, используемый узлом памяти, идентичен используемому узлом процессора.

Метод доступа к памяти в модели ADAM заключается в том, чтобы сначала отправить начальную возможность доступа узлу памяти, а затем отправлять только смещения к этой возможности. Если другая возможность видна приходящей из уже инициализированной пары идентификатор контекста/очередь, это интерпретируется как повторная инициализация возможности доступа для этой пары. Обратите внимание, что пары идентификатор контекста/очередь уникальны во всей машине по замыслу. Поскольку потоки собираются сборщиком мусора или отображения памяти явно уничтожаются, возможности доступа удаляются из таблицы доступа.

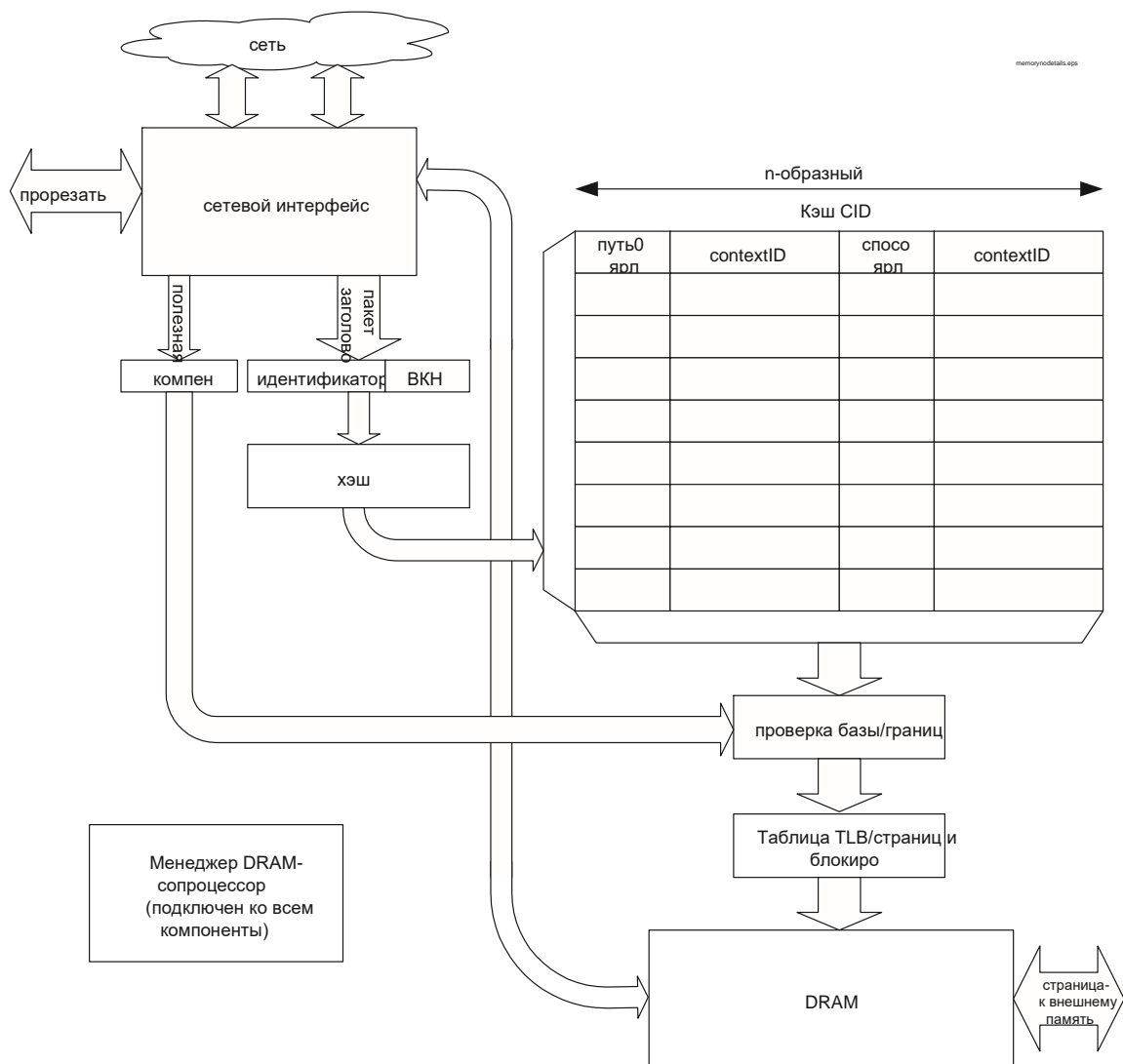


Рисунок 5-5: Высокоуровневая структурная схема узла памяти.

Таблица доступа реализована точно так же, как кэш; см. рисунок 5-5. Индекс и тег в таблице доступа определяются путем хэширования пары идентификатора запрашивающего контекста и номера очереди. Информация об идентификации запрашивающей стороны встраивается в каждый сетевой пакет (включая пакеты, поступающие через сквозной интерфейс). Разработчик может оптимизировать размер и ассоциативность кэша, а также хэш-функцию, чтобы оптимизировать частоту попаданий. В случае коллизии кэша замененные данные не могут быть выброшены, как в кэше; вместо этого данные должны быть удалены в хранилище, которым управляет сопроцессор памяти.

Сопроцессор памяти также отвечает за управление атомарными транзакциями. Когда поток выполняет инструкцию EXCH, пакет отправляется в узел памяти, который отмечает

возможность доступа как атомарную. В следующий раз, когда поток пытается получить доступ к очереди, которая отображается в очередь EXCH, поток (фактически, узел процессора, на котором выполняется поток) согласовывает блокировку возможности и выполняет атомарный обмен. Одной из обязанностей сопроцессора памяти является координация этой функции блокировки.

Адресное пространство, выделенное для каждого узла памяти, намного больше, чем реализованная память в узле памяти; таким образом, предполагается, что каждый узел памяти имеет доступ к более медленному, но очень большому резервному хранилищу, будь то дисковод или обычная DRAM с резервным дисководом. Подкачка выполняется обычным способом, и сопроцессор памяти также отвечает за управление подкачкой.

Симулятор системы ADAM реализует узел памяти как массив с равномерно  
сосредоточенными  
средняя задержка доступа.

## 5.4 Физический дизайн

Я описываю здесь, как может выглядеть физическая реализация Q-машины. Это упражнение является важным шагом в обосновании параметров симулятора в некотором подобии реальности; читателям предлагается пропустить этот раздел, если их мало интересует физическое проектирование.

### 5.4.1 Предположения о технологиях

Прежде чем углубляться в детали физической реализации Q-Machine, я суммирую свои основные технологические предположения. Я предполагаю, что окончательная реализация Q-Machine будет представлять собой машину среднего или большого размера, состоящую из массива чипов процессора плитки. Ожидается, что количество узлов, представленных всем массивом, будет находиться в диапазоне от 1000 для настольной машины до 1 000 000 для суперкомпьютера размером с комнату. Я также предполагаю наличие питания от розетки и, возможно, схему жидкостного охлаждения, использующую микроканалы [Tuc84], чтобы обеспечить максимальный тепловой бюджет не менее

Параметр	Ценить
Литография	50 нм

Длина ворот	30 нм
Слои металла	10 минимум
Короткий шаг провода	100 нм [CI00b]
Короткий провод, максимальный пробег	300 м [CI00b]
Размер чипа (производство)	400 мм
Размер чипа (максимальный)	572 мм
Логическая плотность, автокомпоновка ASIC	400-800 Мтранзисторов/см
Плотность SRAM, высокая производительность	1423 Мтранзисторов/см или 237 Мбит/см
Максимальный размер кэша SRAM при времени доступа 0,3 нс	4 КБ [АНКВ00]
Максимальный размер кэша SRAM при времени доступа 0,5 нс	100 КБ [АНКВ00]
Максимальный размер кэша SRAM при времени доступа 1,0 нс	1000 КБ [АНКВ00]
Ожидаемое соотношение памяти и логики	9:1 [CI00c]
Размер ячейки DRAM, оптимизированный	0,0064 м , или 15,6 Гбит/см
Тактовая частота, ASIC (кросс-чип)	1,5 ГГц
Тактовая частота, локальная	10 ГГц
Тактовая частота, 16FO4 задержки на такт	3,5 ГГц [АНКВ00]
Достижимая область чипа за 1 нс (задержки 16FO4)	около 10% [АНКВ00]
Доступны сигнальные площадки ввода/вывода	2700
Скорость передачи сигналов ДСП	3.1 ГГц
Дефекты ASIC D , D/m (выход 65%)	787
Стоимость при внедрении с использованием высокопроизводительных (большой объем встроенной памяти) Модель ЦП	3.8центов/транзистор
Количество слоев кремния	Не менее 2
Шаг соединения между слоями	Эквивалент металла высшего уровня

Таблица 5.1: Экстраполированные технологические параметры на 2010 год. Все значения взяты из [CI00a], если не указано иное.

1000 Вт на чип (фактическое потребление, как предполагается, намного меньше этого). Я также предполагаю, что реализация будет завершена около 2010 года, плюс-минус пара лет. Одним из наиболее важных аспектов ADAM является то, что он может использовать предстоящий более высокий уровень интеграции, справляясь с задержками проводов, сложностью и проблемами выхода годных, которые обычно ожидаются как проблемы с

технологическим процессом уровня 2010 года, сохраняя при этом обратную совместимость с реализациями ADAM, созданными сегодня. Таблица 5.1 суммирует технологии, которые могут быть доступны в 2010 году.

Обратите внимание, что данные [АНКВ00] основаны на дорожной карте Ассоциации полупроводниковой промышленности (SIA) 1999 года, которая предполагает площадь чипа около 800 мм<sup>2</sup> при 50 нм узле, тогда как все остальные данные взяты из обновленной дорожной карты SIA 2000 года. Доступность 3-D CMOS-процесса не рассматривается дорожной картой SIA, но ряд компаний и исследовательских лабораторий показали многообещающие результаты, такие как Matrix Semiconductor и MIT Lincoln Labs. Matrix Semiconductor продемонстрировала многоуровневые кремниевые устройства, которые были изготовлены по процессу TSMC. Их основной подход заключается в нанесении тонких пленок аморфного кремния на планарные диэлектрические слои, а затем отжиге кремния в кристаллы, достаточно большие для формирования транзисторов. Их подход не обязательно обеспечивает высокопроизводительную логику, но он дает надежды на высокоплотную память. [Sem] Подход MIT Lincoln Labs, с другой стороны, может обеспечить по крайней мере два слоя высокопроизводительной логики. Их подход связывает две пластины или чипа SOI CMOS лицом к лицу с помощью гидрофильной сварки при комнатной температуре. Чтобы создать больше слоев логики, одну или обе соединенные пластины можно утончить с помощью процесса травления и/или химико-механической полировки (CMP), используя слой SiO в качестве ограничителя травления. Другой слой логики можно гидрофильно соединить, и процесс повторить. [LBCF 00]

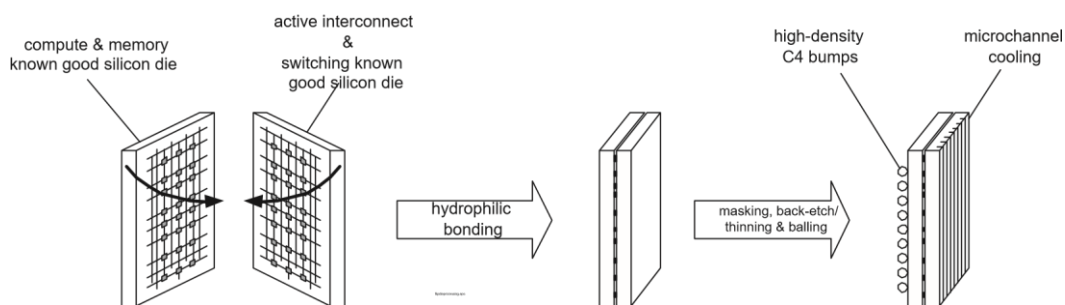


Рисунок 5-6: Упаковка и интеграция двухслойного кремниевого высокопроизводительного чипа мультипроцессора.

Этот, по общему признанию, нечеткий взгляд в облачный хрустальный шар будущего формирует основу для некоторых констант, связанных с реализацией Q-машины. Важно



повторить, что ADAM не полагается на то, что какая-либо из этих технологий будет реализована; можно было бы реализовать ADAM в современные технологии.

#### **5.4.2 Описание дизайна**

Физическая конструкция Q-Machine использует высокопроизводительный двухслойный кремниевый процесс, как показано на рисунке 5-6. Один слой предназначен для процессорных узлов, а другой слой предназначен для активной коммутационной сети. Каждый слой может быть независимо протестирован перед интеграцией с помощью встроенного самотестирования (BIST) и зондирования пластины, чтобы помочь повысить выход систем.

Такое разделение проекта на сетевой и процессорный уровни имеет несколько преимуществ. Отдавая целый уровень активной коммутационной сети, межсоединение может использовать более толстые, широко разнесенные дифференциальные провода, а размещение буфера менее ограничено. Кроме того, межсоединительный уровень содержит все маршрутизаторы и коммутаторы. Наконец, межсоединительный уровень полностью универсален: пользователь может повторно использовать межсоединительный уровень в нескольких проектах и включать пользовательские узлы на процессорном уровне. Архитектура, которая использует этот вид реконфигурируемости, описана в [CCN 00]. Очевидное преимущество для процессорного уровня заключается в том, что он свободен от накладных расходов на сетевую проводку и буферизацию, и, таким образом, имеет меньше ограничений по размеру, компоновке и размещению узлов. Схема того, как может выглядеть сетевой уровень, представлена на рисунке 5-7. Обсуждение топологии сети, выбранной для этой реализации, можно найти в разделе C.3.

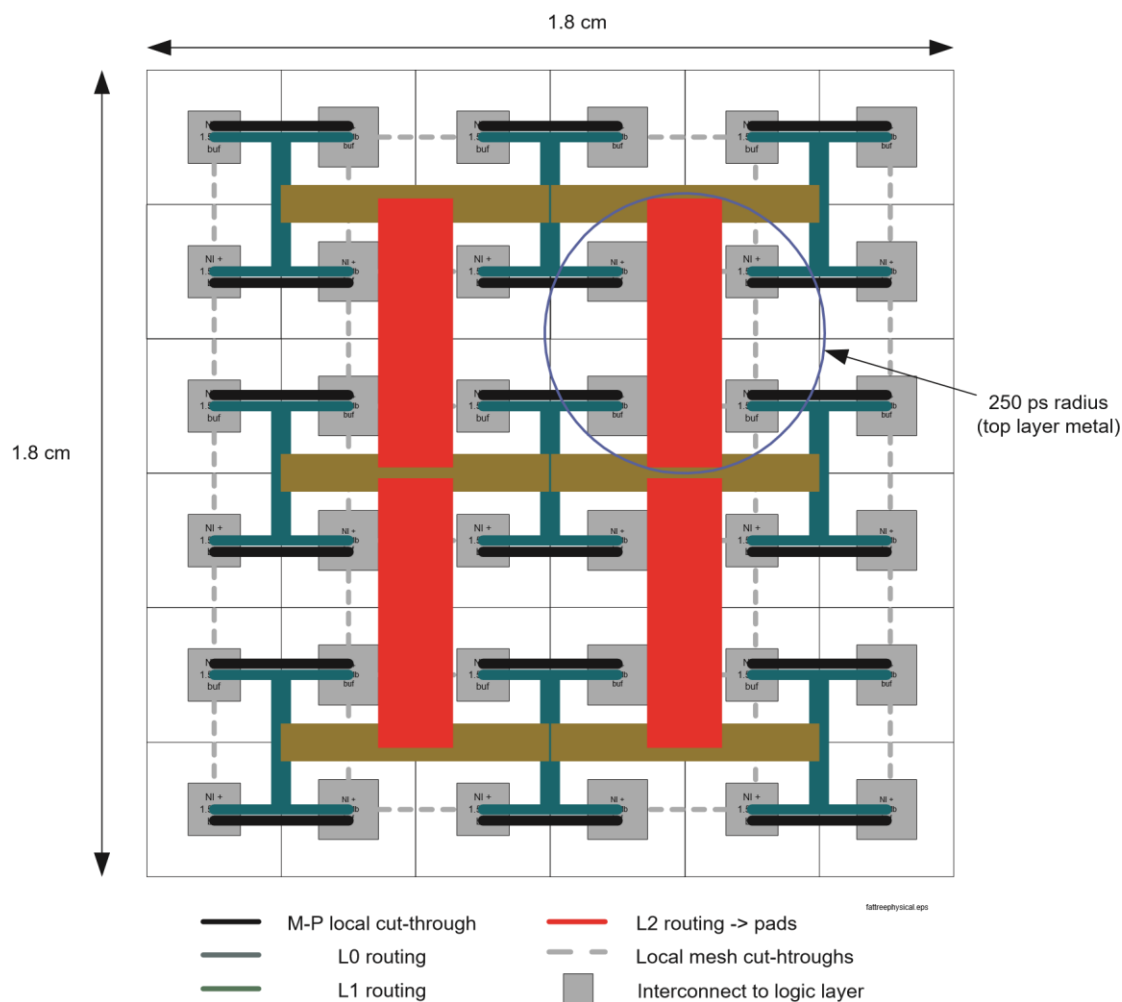


Рисунок 5-7: Рисунок компоновки сетевого уровня.

Слой процессора для этой реализации выбран в виде простого формата плитки. Каждая плитка блока состоит из узла памяти и узла процессора (архитектурная блок-схема узла процессора представлена на рисунке 5-2 с описанием в разделе 5.3). Узел памяти и узел процессора размещены в виде торов вокруг сетевого интерфейса (NI). Такое тороидальное расположение вокруг сетевого интерфейса помогает минимизировать наихудшее расстояние любого из более медленных проводов, используемых на уровне процессора, до более быстрого соединения на сетевом уровне. Обзор того, как может выглядеть полностью плиточный узел, представлен на рисунке 5-9.

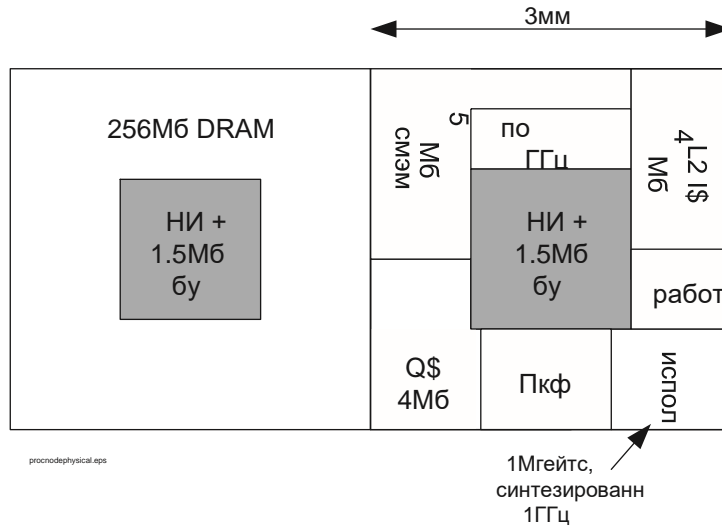


Рисунок 5-8: Гипотетическая компоновка одного процессорного узла.

Ожидаемая тактовая частота одного процессорного узла составляет 1 ГГц. Это число получено путем рассмотрения радиуса связи в течение интервала 1000 пс и предполагаемого периода тактовой частоты для логики, построенной с использованием правила проектирования 64 уровней инвертора FO4 на узле 35 нм процесса. [АНКВ00] Относительно мягкие критерии 64 уровней инвертора FO4 были выбраны для того, чтобы позволить выполнить проектирование процессора с меньшим количеством этапов конвейера и в первую очередь синтезированной методологией проектирования Verilog с несколькими хорошо выбранными вручную оптимизированными блоками (такими как умножители, сумматоры и баррельные сдвигатели). Проще говоря, предположения о физической реализации этой архитектуры в моей диссертации сохраняются очень консервативными, чтобы помочь компенсировать их чрезвычайно спекулятивный характер. Кроме того, фактическая производительность механизма миграции в моей диссертации всегда указывается в терминах сетевых циклов и сравнивается с другими реализациями путем нормализации указанных времен к тактовым циклам. Нормализация к тактовым циклам помогает исключить технологические предположения. Наконец, поскольку производительность механизма миграции в этой архитектуре доминирует над производительностью сети, фактическая тактовая частота процессора может быть намного выше и иметь незначительное влияние на результаты этой диссертации. Подробные предположения о сетевом интерфейсе приведены в приложении С. Краткое резюме

заключается в том, что сетевой интерфейс должен быть способен отправлять, в худшем случае, один флит каждые 500 пс до 1 нс.

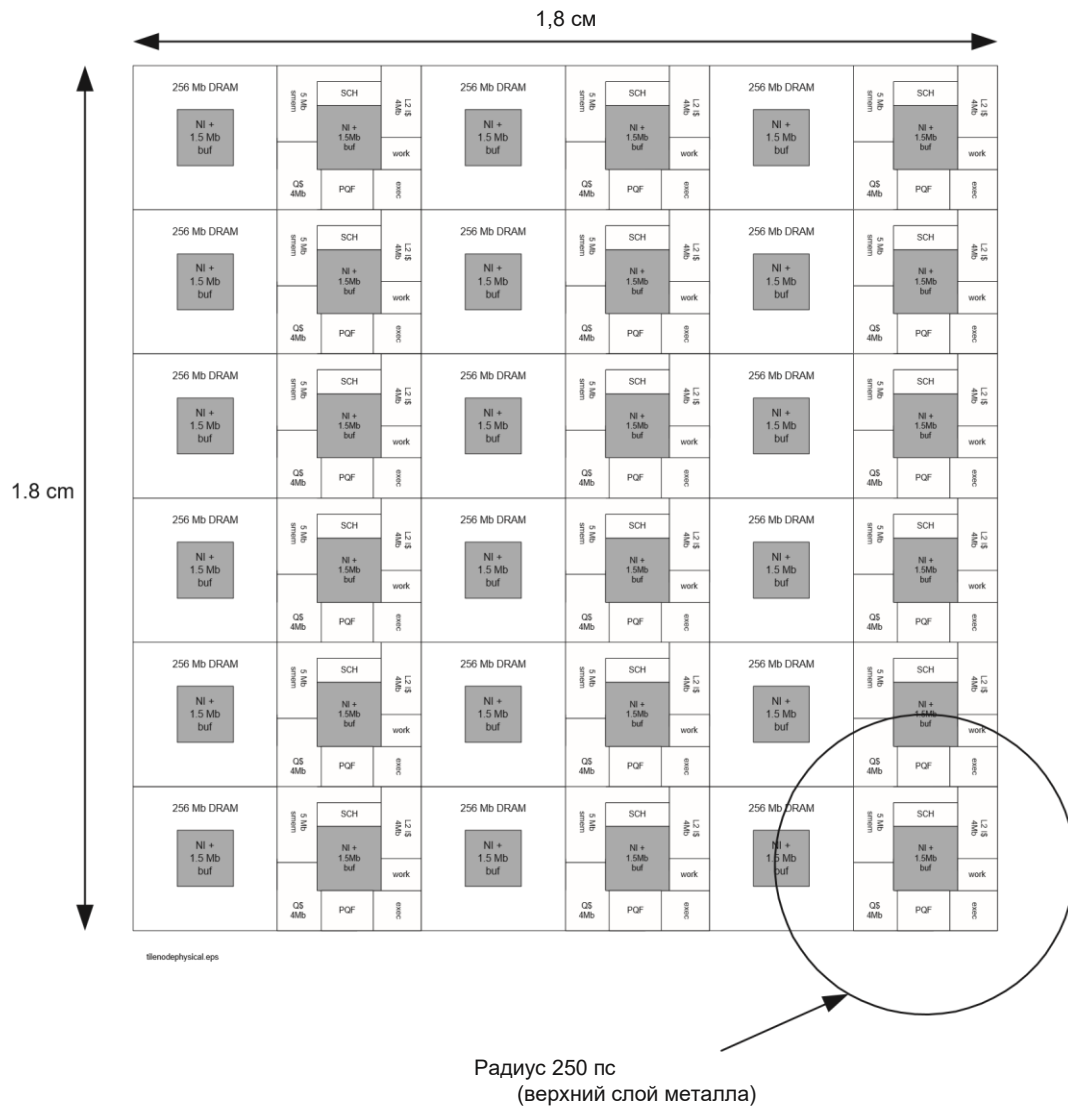


Рисунок 5-9: Гипотетическая компоновка чипа процессора плитки.

## Глава 6

# Машина и миграция

## Характеристика

42,7% всех статистических данных составляются на месте.

— *Достопочтенный У. Ричард Уолтон-старший.*

В последнем разделе описывается быстрый, малонакладный механизм перемещения данных и потоков в рамках реализации Q-Machine ADAM. В этом разделе суммируются основные характеристики производительности Q-Machine и представляются результаты и анализ нескольких бенчмарков.

### 6.1 Основные результаты производительности Q-Machine

В этом разделе представлены некоторые основные характеристики производительности Q-Machine и ее сети, реализованные Adam System Simulator. Предполагается, что все инструкции выполняются со скоростью одна за цикл, пока удовлетворяются ее зависимости. Инструкция, которая была запланирована, но не имеет зависимости, приводит к вставке одноциклового пузыря в поток выполнения. Будущие реализации могут спроектировать PQF для взаимодействия с планировщиком таким образом, чтобы этот пузырек был устранен, однако этот симулятор поколения был написан для максимального упрощения реализации процессорного узла. Как упоминалось ранее, будущие реализации также могут повысить производительность процессорного узла, добавив в ядро неисправную, суперскалярную проблему или SMT. Последние две функции избегаются, поскольку они потребовали бы большего количества портов файла очереди; первая потребовала бы ассоциативного поиска в окне ожидающей неисправной проблемы в потоке всякий раз, когда фрагмент данных поступает из сетевого интерфейса.

Adam System Simulator (ASS), используемый для получения всех результатов для моей диссертации, реализует полный идемпотентный сетевой протокол, ответственный за источник, описанный в разделе С.2, и моделирует сеть с точностью до цикла. Топология сети представляет собой случайное толстое дерево с основанием 4 и расширением 2, — масштабирование полосы пропускания на уровне дерева; вся проводка однонаправленная точка-точка. Каждый узел процессора или памяти имеет четыре порта в сети: два входных и два выходных. Моделируемые узлы процессора также учитывают механизмы подкачки, необходимые для файла очереди именованных состояний (как описано в приложении С). Симулятор также учитывает простои из-за ограничений полосы пропускания памяти. Задержка сетевого интерфейса во всех режимах маршрутизации (cut-through, loopback и off-node routing) также учитывается симулятором. Симулятор был написан на Java и достигает пиковой скорости моделирования около 20 кциклов/с в совокупности на системе с двумя процессорами Athlon XP 1900+. Скриншот ASS можно увидеть на рисунке 6-1.

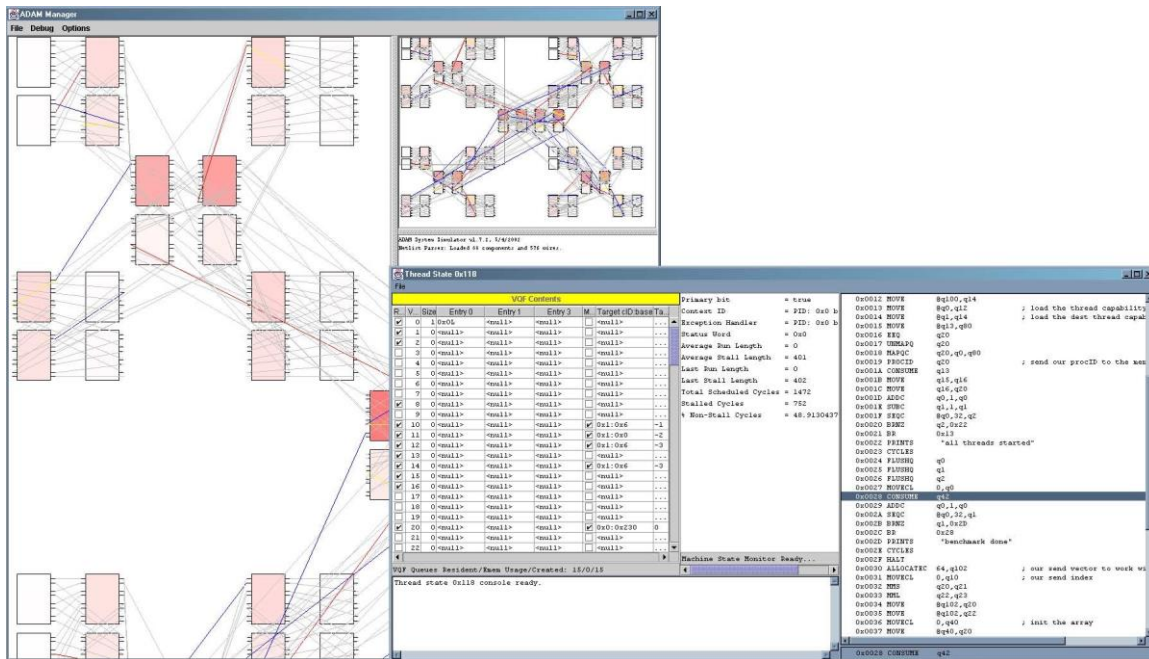


Рисунок 6-1: Скриншот ASS, выполняющего 64-узловой векторный обратный регрессионный тест. Слева — обзор машины; справа — окно отладчика потоков.

### 6.1.1 Память Performance

Эти результаты суммируют время доступа к узлу процессора, необходимому для предпочтительного узла памяти. Все следующие результаты относятся к незагруженной машине, на которой выполняется только тестовый код.

**Задержка распределения:** 10 циклов от выполнения инструкции ALLOCATE до выдачи зависимой от нее инструкции. Алгоритм распределения в симуляторе прост; он просто увеличивает указатель распределения и возвращает возможность требуемого размера.

**Загрузка из локальной памяти:** 7 циклов от выполнения инструкции MOVE, которая отправляет адрес загрузки, до выдачи инструкции, зависящей от результата загрузки. Разбивка времени загрузки: 1 цикл от ядра процессора NI; 1 цикл от NI cut-through port preferred memory node; 2 цикла для выполнения доступа к памяти; 1 цикл memory memory node NI; 1 цикл от memory node NI процессор; 1 цикл для повторного планирования и повторной выдачи зависимое обучение.

**Сохранить в локальной памяти:** 6 циклов от выполнения инструкции MOVE, которая удовлетворяет атомарному адресу и кортежу данных, до выдачи зависимой инструкции. Разбивка задержки сохранения похожа на загрузку, за исключением того, что в памяти тратится только 1 цикл, поскольку возврат подтверждения сохранения может перекрываться сохранением.

Эти цифры являются консервативными для целевого технологического процесса; в бюджете предусмотрен один цикл в каждом направлении для задержек проводов из-за предполагаемого расстояния между памятью и процессором.

### 6.1.2 Базовая производительность сетевых операций

Общая формула для задержки пакета связи между потоками ( ) является:

(6.1)

где — накладные расходы сетевого интерфейса узла процессора, равные 2 циклам; — время, необходимое для прохождения маршрутизатора, равное 3 циклам; — глубина маршрута, равная числу уровней вверх по дереву, которые должен пройти пакет; — длина пакета данных, равная 4 для короткого пакета данных.

Следующие тесты были проведены на незагруженной 16-процессорной модели; эти тесты подтверждают справедливость уравнения 6.1.

**Задержка обратной связи:** Задержка обратной связи — это время, необходимое потоку для связи с другим потоком на том же узле процессора. Это время составляет 4 цикла от выполнения производителя до выдачи инструкции-потребителя при минимальной загрузке планировщика. Разбивка 4 циклов следующая: 1 цикл от ядра процессора NI; 1 цикл для идентификации и обработки обратной связи; 1 цикл от порта записи NI PQF; 1 цикл для повторного планирования и повторной выдачи зависимой инструкции. Задержки при реальных рабочих нагрузках обычно включают некоторое количество времени, потраченного на обслуживание других потоков в рабочей очереди.

**Время от потока к потоку:** Задержка отправки одного фрагмента данных на узел, который находится на расстоянии одного маршрута вверх, составляет 14 циклов, не считая времени выпуска и выполнения потоков производителя и потребителя. Таким образом, односторонняя задержка до ближайшего соседа составляет примерно 14 циклов. Распределение задержки следующее (получено путем измерений):

- 2 цикла после выполнения производителя для отправки пакета в NI
- 7 циклов по сети (первый контакт с сетью до первого контакта в точке назначения с задержкой) = 3 цикла/маршрутизатор + 1 цикл провода между маршрутизаторами
- 2 цикла для «догонялки» хвоста пакета
- 3 цикла по составлению и выдаче потребительской инструкции

**Задержка доступа к удаленной загрузке, полный диаметр на машине с 16 узлами:** 55 циклов туда и обратно  
задержка



## 6.2 Эффективность миграции и контроль миграции: простые случаи

В этом разделе представлены результаты применения механизмов миграции к двум простым случаям: два потока, взаимодействующих исключительно друг с другом, и поток и память, взаимодействующие исключительно друг с другом. Также выполняется краткий формальный анализ для определения оптимальных всезнающих и оптимальных онлайн-алгоритмов для управления миграцией в этих случаях.

### 6.2.1 Тест производительности двух потоков

Этот бенчмарк используется для определения накладных расходов на миграцию потоков. Строится плотный цикл зависимых операций между двумя потоками; разница во времени на цикл сообщений во время миграции потоков и во время нормальной работы является накладными расходами на миграцию. В частности, два потока взаимодействуют исключительно друг с другом. Каждый поток инициализируется уникальным токеном; токены обмениваются между потоками и увеличиваются в течение 32 итераций. На пятой итерации вручную

Выдается инструкция `MIGRATE`, которая заставляет один поток мигрировать к своему партнеру. Использование вручную вызываемой миграции позволяет лучше контролировать процесс бенчмаркинга. Снижения накладных расходов процессора при ручном управлении миграцией не происходит, поскольку эта задача обычно выполняется сопроцессором планирования и профилирования. Диаграмма шаблона связи представлена на рисунке 6-2, а код для этого синтетического бенчмарка — на рисунке 6-3.

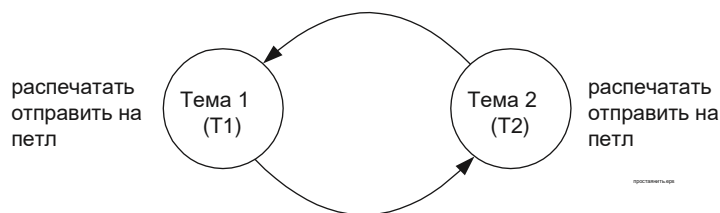


Рисунок 6-2: Синтетический бенчмарк с двумя потоками. Связь происходит по дугам; зависимость данных принудительно устанавливается путем печати входящих данных.

## Результаты двухпоточного теста

Тест двух потоков с миграцией был запущен в пяти случаях, в которых менялась начальная позиция потоков. Эти испытания сравнивались с тестом двух потоков без миграции в тех же пяти начальных позициях. Тест дал следующие результаты:

Измеренные временные затраты на легкую миграцию на расстояние двух восходящих маршрутов составляют 66 циклов. Временные затраты вычисляются как время, добавленное к результату одной итерации по сравнению с неперенесенным случаем.

Измеренные временные затраты на миграцию с большим весом на расстояние двух восходящих маршрутов составляют 78 циклов; миграция с большим весом была вызвана настройкой внутреннего порога принятия решения о тяжелом/легком уровне симулятора. Ускорение эталонного теста линейно масштабируется с расстоянием миграции (рисунок 6-4). Ускорения больше в реальной реализации системы, поскольку среда моделирования предполагает, что задержки проводов между уровнями дерева постоянны, независимо от размера дерева. Другими словами, реальная реализация будет иметь большую задержку проводов, особенно в большой реализации, и влияние миграции будет больше.

Случай нулевого расстояния на рисунке 6-4 показывает производительность ниже единицы, поскольку существуют небольшие накладные расходы в размере 14 циклов для выполнения команды миграции вручную, даже если она не перемещает поток.

Расчетная системная задержка заполнения кэша L2 на Pentium 4 составляет около 175 нс (что составляет 140

800 МГц циклов Direct-RAMBUS) [CJDM01]. Время миграции потоков в моей архитектуре, таким образом,

основной:

```
MOVECC0, d100          ; создать один локальный поток
MOVECC 4, q101          ; создать один поток на расстоянии
                        4
SPAWNC q100, поток1, q0
SPAWNC q101, поток2, q1
MARQC q10, q0, @q0
MARQC q11, q0, @q1
ДВИГАТЬСЯ @q0, q11      ; поток 1, встречаем поток 2
ДВИГАТЬСЯ @q1, q10      ; поток 2, встречаем поток 1
ОСТАНОВИТЬ ; моя работа выполнена thread1:
MOVECL0, q10
MARQC q20, q1, @q0
```

петля1:

```
ДВИГАТЬСЯ @q10, q20
ПЕЧАТЬQX q1
SEQC @q10, 0x20, q30
SEQC @q10, 0x5, q40 ; этот сегмент используется для
управления, когда
```

```

АДЦК q10, 1, q10          ; миграция происходит во время тестирования
БРЗ    q40, буp1
ПРОЦИД q41
MOVECL 2, q42
МИГРАЦИЯ q41, @q0
буp1:
    БРЗ    q30, петля1
    ЦИКЛЫ          ; CYCLES выводит текущее количество циклов
    ОСТАНОВИТЬ      ; доступно только в среде моделирования
поток2:
    MOVECL 0x100, q10
    MAPQS q20, q1, @q0
петля2:
    ДВИГАТЬСЯ @q10, q20
    ПЕЧАТЬQX q1
    SEQC @q10, 0x120, q30
    АДЦК q10, 1, q10
    БРЗ    q30, петля2
    ОСТАНОВКА
    ЦИКЛОВ

```

---

Рисунок 6-3: Код, используемый для двухпоточного теста.  
выгодно отличается от заполнения кэша L2 на обычном современном процессоре.

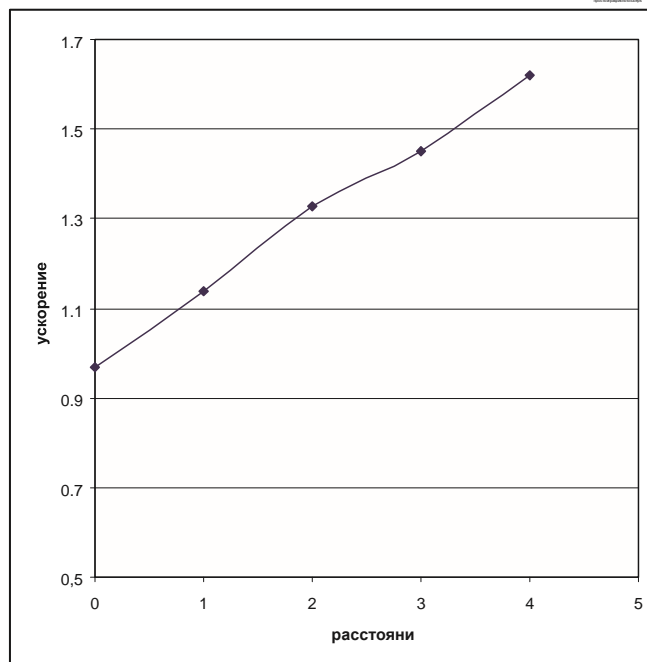


Рисунок 6-4: Измеренное ускорение в зависимости от расстояния миграции для теста  
Two Threads.

## Анализ

Теперь я выведу онлайн-алгоритм для принятия решений о миграции в двухпоточном сценарии, а также определю, сколько итераций цикла должно произойти, чтобы окупить стоимость миграции. Формально говоря, потоки взаимодействуют с помощью последовательности сообщений. Для любого заданного  $\alpha$ , я выведу алгоритм миграции,  $ALGTT$ , и оценю его конкурентоспособность по отношению к оптимальному алгоритму,  $OPTTT$ .

В случае микротеста Two Threads последовательность состоит из сообщений, каждое из которых вносит частичную стоимость в зависимости от расстояния маршрутизации:

(6.2)

Обозначим стоимость алгоритма – время, необходимое для выполнения заданного – как  $ALG$ . Если стоимость перемещения потока равна  $\alpha$ , то наши алгоритмы определены для:

$OPTTT$ : Если  $OPTTT$  перенесет поток  $l$  из  $k$  до первой итерации.

$ALGTT$ : Если на итерации  $i$ ,  $ALGTT$  перенесет поток  $l$  из  $k$  до первой итерации.

В настоящее время выведена конкурентоспособность  $ALGTT$  и  $OPTTT$ .

**Теорема.**  $ALGTT$  в худшем случае 2-конкурентноспособен с  $OPTTT$ .

**ДОКАЗАТЕЛЬСТВО.**

По результатам проверки,  $OPTTT = ALGTT$  для  $OPTTT \leq OPTTT$ . Давайте

определить значение, где  $OPTTT$  перестает быть равным  $ALGTT$  как точка эквивалентности.

В случаях, когда  $\alpha > 1$ , конкурентное отношение  $ALGTT$  к  $OPTTT$  равно

(6.3)

Очевидно, что худшим вариантом было бы, если бы, потому что  $ALGTT$  заплатит за  $\alpha$  и никогда не сможет амортизировать его стоимость. На этом этапе конкурентное соотношение просто

(6.4)

где

Таким образом, как ,АЛГТТ ОПТТТ.

**Теорема.** *ALGTT — оптимальный онлайн-алгоритм для микротеста Two Thread.*

**ДОКАЗАТЕЛЬСТВО.**

При сравнении ALGTT с другим алгоритмом, ALG, следует учитывать два случая:

**МИГРАЦИЯ РАНЬШЕ.** В случае, если ALG мигрирует раньше, чем ALGTT, худшим вариантом производительности для ALG будет последовательность, которая заканчивается прямо на пороге принятия решения ALG. Конкурентное отношение в этом случае будет

(6.5)

Видно, что эта функция монотонно возрастает при уменьшении значений (рисунок 6-5); следовательно, ALG не может иметь более низкий коэффициент конкурентоспособности, чем ALGTT.

**МИГРАЦИЯ ПОЗЖЕ.** В случае, если ALG мигрирует позже, чем ALGTT, худшим вариантом производительности для ALG снова будет последовательность, которая заканчивается прямо на пороге принятия решения ALG. В этом случае конкурентное отношение будет

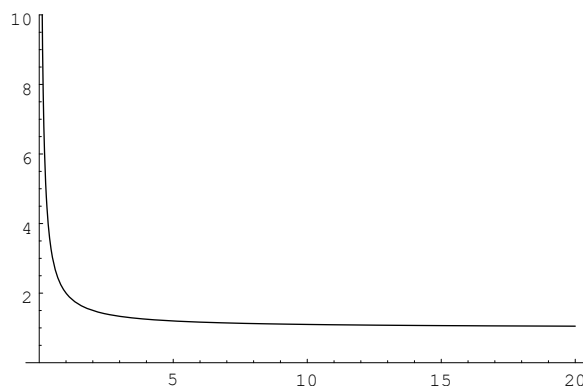


Рисунок 6-5: Форма кривой\_\_\_\_\_.

(6.6)

где  $\alpha$ , и  $\beta$  является точкой эквивалентности, как было определено ранее. Как решение

точка ALG увеличивается за пределы  $\beta$ , числитель уравнения 6.3 растет медленнее, чем числитель уравнения 6.6, так как  $\beta < \alpha$ . Поскольку знаменатели одинаковы, ALG не может быть меньше ALGTT.

Теперь я определю кривую для точки эквивалентности,  $\alpha$ , в зависимости от различных накладных расходов на миграцию. Точка,  $\alpha$ , представляет, где стоимость миграции потока амортизируется экономией времени коммуникации потока. Чтобы определить это, я выведу выражение для времени доставки сообщения,  $\alpha$ . Общая формула для задержки маршрутизации в реализации Q-Machine:

(6.7)

где  $\alpha$  — вклад задержки маршрутизаторов для одного уровня дерева, а  $\beta$  — расстояние маршрутизации, т. е. количество уровней дерева, охватываемых маршрутом. В Q-машине,  $\alpha$ . Напомним, что точка эквивалентности определяется как

(6.8)

Переписывание дает

(6.9) Подставляя в 6.7,

(6.10)

Учитывая уравнение 6.10, мы можем создать набор кривых, показывающих, сколько итераций требуется для амортизации стоимости миграции для различных затрат на миграцию (рисунок 6-6). Стоимость миграции потока,  $\alpha$ , предполагается постоянной для каждой из этих кривых, что является разумным предположением, поскольку для этих графиков предполагается тяжеловесный механизм миграции потока. При рассмотрении кривых становится очевидным, что стоимость миграции быстро амортизируется, даже для сетей скромного размера с постоянной задержкой проводов между уровнями дерева. Миграция выглядит еще более привлекательно в реалистичном сценарии, где задержки

проводов растут в лучшем случае как квадратный корень или кубический корень числа узлов в машине.

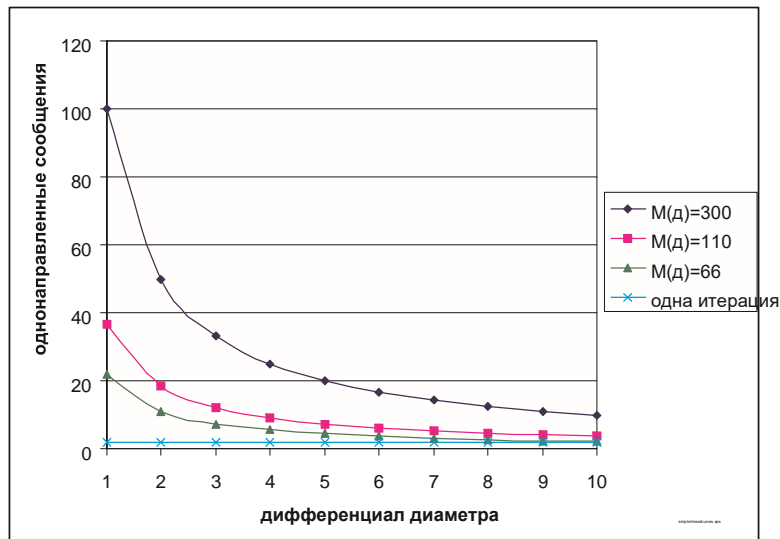


Рисунок 6-6: Длина последовательности сообщений, необходимая для амортизации различных накладных расходов на миграцию ( ). На графике также отмечен базовый уровень — два сообщения на итерацию для двухпоточного теста.

### 6.2.2 Тест потоков и памяти

Тест потока и памяти используется для определения накладных расходов на миграцию данных, аналогично тесту двух потоков. В тесте потока и памяти один поток взаимодействует исключительно с одним фрагментом памяти из 16 или 128 слов через сопоставление обмена. Одно местоположение в памяти увеличивается 32 раза удаленным потоком с помощью механизма обмена; на пятой итерации данные принудительно перемещаются в поток. Этот сценарий похож на случай двух потоков, за исключением того, что для нелокальной памяти вводится дополнительный уровень косвенности из-за механизма указателя локатора данных. Код для этого синтетического теста можно найти на рисунке 6-8, а диаграмму шаблона связи можно найти на рисунке 6-7.

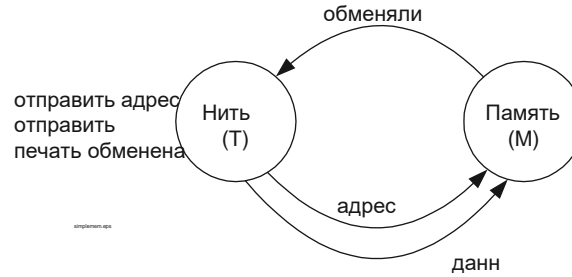


Рисунок 6-7: Синтетический бенчмарк потоков и памяти. Связь происходит по дугам; зависимость данных принудительно устанавливается путем печати входящих данных.

### Результаты тестов потоков и памяти

Тест Thread-Memory с миграцией был запущен в пяти случаях, которые варьировали начальную позицию памяти. Эти испытания сравнивались с тестом Thread-Memory без миграции в тех же пяти начальных позициях. Результаты теста суммированы на рисунках 6-9 и 6-10.

На рисунке 6-10 показано количество времени на итерацию в зависимости от количества итераций для миграции, принудительно вызванной на пятом цикле для случаев из 16 и 128 слов. Ключевые точки данных помечены статусом миграции на этой итерации. Объем накладных расходов на миграцию, который фактически испытывает система, зависит от относительного времени запроса на миграцию и входящих запросов памяти. В случае из 16 слов время таково, что фактически нет накладных расходов из-за замораживания запроса памяти и конкуренции во время процесса миграции; поскольку это единственный шаг, который приводит к замедлению относительно случая без миграции, миграция небольших объектов памяти практически бесплатна. Однако накладные расходы на миграцию масштабируются линейно с размером перемещаемых данных, и в конечном итоге процесс замораживания и перемещения возможности добавляет немалые накладные расходы, как можно увидеть в случае перемещения возможности из 128 слов.

Накладные расходы и характеристики запроса сообщений для случая Thread-Memory аналогичны случаю Two Thread; миграцию памяти можно рассматривать как миграцию потоков, но быстрее. Следовательно, алгоритмы и анализ из предыдущего раздела применимы здесь.

основной:

```

MOVECC 2, q100          ; создать один поток на расстоянии
                        2
MOVECC0, d101           ; выделить память

```



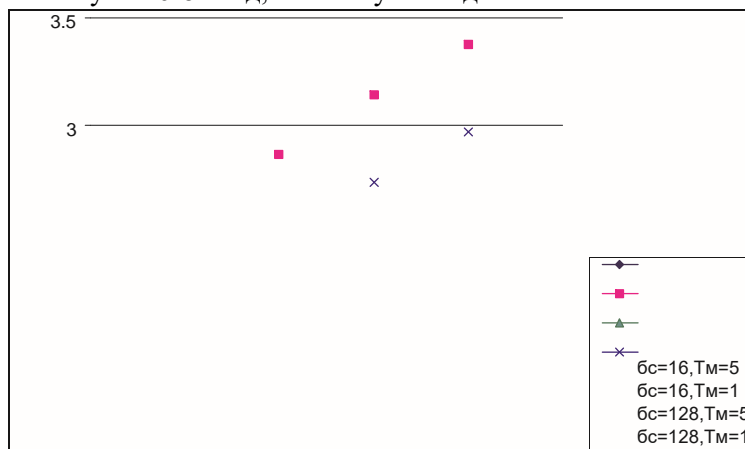
```

        SPAWNC q100, поток1, q0
        АЛЛОКАТЕК q101, 16, q1
        MAPQC q10, q0, @q0
        ДВИГАТЬСЯ @q1, q10          ; нить, встречай свою память
        ПРОЦИД q5
        ДВИГАТЬСЯ к5, к10           ; нить, встретиться со мной
        МИГРАЦИЯ @q1, q20
        ; ИСПОЛЬЗОВАНИЕ             ; замените migrate, чтобы отключить mig.
        q20 PRINTS
        "миграция"
        ОСТАНОВИТЬ                  ; моя работа сделана
поток1:
        EXCH Q20, Q21, Q22          ; объявляем очереди обмена
        MOVECLO, q10
        ПЕРЕМЕЩЕНИЕ q0, q1         ; сохранить наши возможности в q1
        ДВИГАТЬСЯ @q1, q20          ; инициализируем кортеж обмена
        MAPQC q6, q20, q0; вернитесь к нашему вызывающему
петля1:                                объекту...
        MOVECLO, q20                ; всегда используйте адрес 0 для этого
                                    теста

        ДВИГАТЬСЯ @q10, q21
        ПЕЧАТЬQX q22
        SEQC @q10, 0x20, q30
        SEQC @q10, 0x5, q40 ; этот сегмент используется для
        управления, когда
        АДЦК q10, 1, q10 ; миграция происходит во время тестирования
        БРЗ q40, буp1
        ПРОЦИД q5
        ХОД q5, q6                  ; отправить пакет нашему абоненту...
буp1:
        БРЗ q30, петля1
        ЦИКЛЫ                       ; CYCLES выводит текущее количество циклов
        ОСТАНОВИТЬ                  ; доступно только в среде моделирования

```

Рисунок 6-8: Код, используемый для теста потоковой памяти.



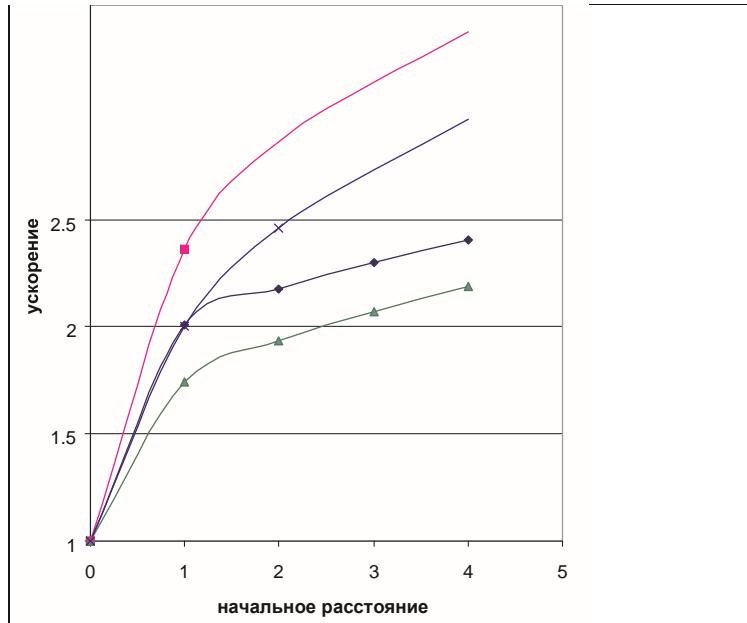


Рисунок 6-9: Ускорение миграции в зависимости от времени принятия решения о миграции и объема памяти в тесте потоков и памяти.

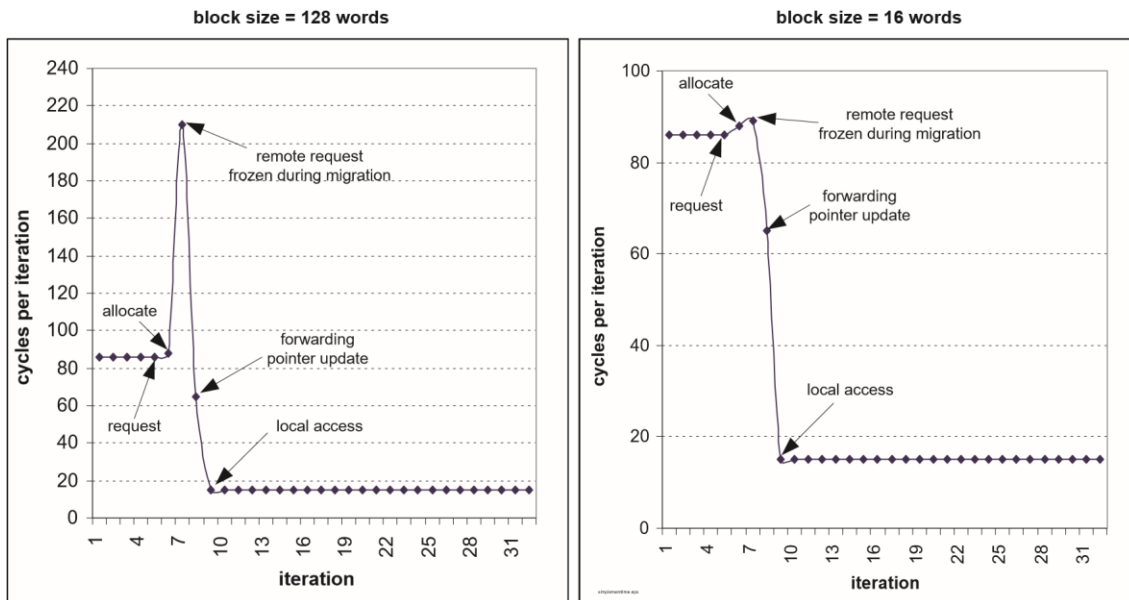


Рисунок 6-10: Циклы на итерацию для теста Thread-Memory. в обоих случаях.

### 6.3 Случаи применения

Теперь я продемонстрирую Q-Machine, на котором запущены некоторые прикладные ядра, закодированные на языке People. Эти приложения — это быстрая сортировка на месте,

потокое матричное умножение и простой гравитационный симулятор N-Body. Приложение быстрой сортировки на месте используется для демонстрации возможностей реализации по балансировке нагрузки. Потокое матричное умножение используется для демонстрации миграции данных с учетом задержки, а гравитационное моделирование N-Body используется для демонстрации миграции потоков с учетом задержки. Чтобы продемонстрировать мою архитектуру на этих ядрах, были реализованы некоторые простые алгоритмы балансировки нагрузки и миграции. Для каждого приложения я кратко расскажу о применяемых методах балансировки нагрузки или миграции, а затем представлю результаты бенчмарка приложения с использованием и без использования динамического управления миграцией.

### **6.3.1 Приложение быстрой сортировки на месте**

Для этого теста на языке People была написана простая быстрая сортировка на месте. Бен Вандивер, создатель People, написал код теста. Реализация быстрой сортировки очень похожа на типичную рекурсивную реализацию, написанную на C или Java. Рисунок 6-11 дает представление о ядре быстрой сортировки. Обратите внимание, что в таком языке, как C или Java, эта рекурсивная реализация имела бы небольшой параллелизм, поскольку каждый рекурсивный вызов `qsort()` вызывается последовательно; однако в People каждый рекурсивный вызов фактически порождает новый поток. Это соглашение о вызовах, порождающих потоки, вводит скрытый параллелизм в код, который может быть обнаружен механизмом балансировки нагрузки.

Схема балансировки нагрузки, реализованная для этого бенчмарка, использует два механизма: `workstealing` и `thread-pushing`. `Workstealing` уже наблюдался в таких системах, как Cilk [BL94] и [RSAU91]; `thread-pushing` — это проблема динамического планирования работы. Обзор алгоритмов и методов динамического планирования работы можно найти в [SHK95] и в [XL97].

Метрикой, используемой для определения загрузки процессора в целях балансировки нагрузки, является время, которое текущий поток провел в ожидании в работающем пуле. Это прямая мера потерянного времени, поскольку планировщик перемещает в работающий пул только те потоки, у которых есть его данные.

зависимости разрешены. Если ожидаемое время, необходимое для миграции потока, перехват работы

полезно, если .

Чтобы реализовать кражу работы, мне нужно было определить и мне пришлось реализовать распределение нагрузки

механизм покрытия. был определен путем профилирования; подробные результаты профилирования миграции можно увидеть на рисунке 6-12. Примечательным наблюдением является то, что время миграции принимает би-

```
int qsort(array[int] arr, int low, int high) {
    если (высокий ==
        низкий) { вернуть
            высокий-низкий;
    } else { int pivot,
        index, temp; int i,j;
        boolean notdone;

        // выбираем pivot index = low
        + rand(high-low); pivot =
        arr[index]; arr[index] =
        arr[low]; arr[low] = pivot;

        // раздел i
        = низкий -
        1; j =
        высокий + 1;
        невыполнено = правда;

        пока (не выполнено)
        { пока (не
            выполнено) { j =
                j - 1;
                notdone = arr[j] > pivot;
            } notdone =
            true; пока
            (notdone) { i =
                i + 1;
                notdone = arr[i] < pivot;
            }
            если (я < j) {
                temp = arr[i]; arr[i] = arr[j]; arr[j] = temp; notdone
                = true;
            } иначе { notdone
                = false; индекс
                = j;
            }
        }

        мембар();

        // рекурсия i =
        qsort(arr,low,index); j =
```

```

    qsort(arr, index+1, high);
    return i + j;
}
}

```

Рисунок 6-11: Метод объекта для теста быстрой сортировки, написанный на языке People.

Модальное распределение при наличии сильно загруженной сети. Это бимодальное распределение является результатом

коллизий пакетов в сети. Другими словами, это ожидаемое время, при условии, что пакет данных миграции будет успешным с первой попытки. В случае, если пакет миграции не будет успешным с первой попытки, можно сократить накладные расходы и немедленно отказаться от миграции этого потока. Это потребовало бы внесения изменений в протокол миграции, чтобы предотвратить запуск двух копий потока, хотя в случае, если подтверждение пакета миграции не было доставлено. Эта схема миграции fail-fast не была реализована для этих симуляций, но была оставлена в качестве упражнения для будущей

работы.

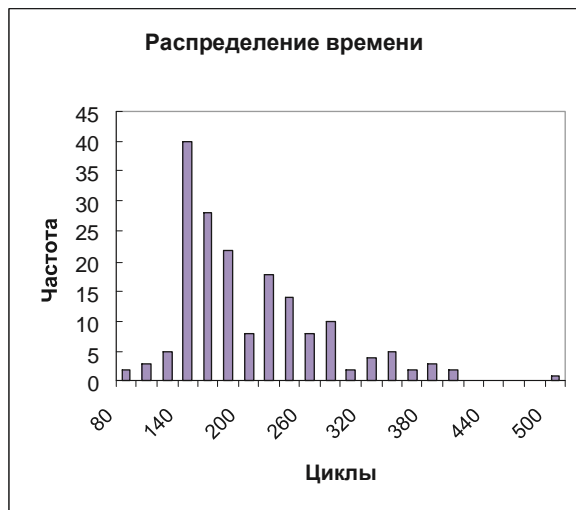


Рисунок 6-12:

Иметь в виду	192
Медиана	170
Режим	125
Стандартное отклонение	72
Минимум	58
Максимум	497
Считать	177

Распределение времени миграции, используемое в тесте быстрой сортировки

Я реализовал механизм обнаружения нагрузки с помощью периодических запросов обнаружения. Период между запросами увеличивается экспоненциально с расстоянием между узлами, поскольку количество требуемых запросов растет экспоненциально с радиусом сети. Базовый период для запросов обнаружения между

соседние узлы настроены на циклы. Этот период был выбран немного больше, чем в попытке обеспечить период выборки, сбалансированный с ожидаемой скоростью изменения в результате миграции. Ответ на запрос обнаружения — это усредненное по времени процессора за последние двадцать событий выполнения потоков.

Данный , я установил номинальный порог кражи на уровне для ближайших соседей. Порог кражи увеличивается линейно с расстоянием между узлами, чтобы компенсировать дополнительные накладные расходы на маршрутизацию при достижении более дальних узлов. Оптимальная скорость увеличения порога кражи с расстоянием, вероятно, нелинейна, но не повлияет на этот бенчмарк, поскольку работы достаточно только для двух узлов.

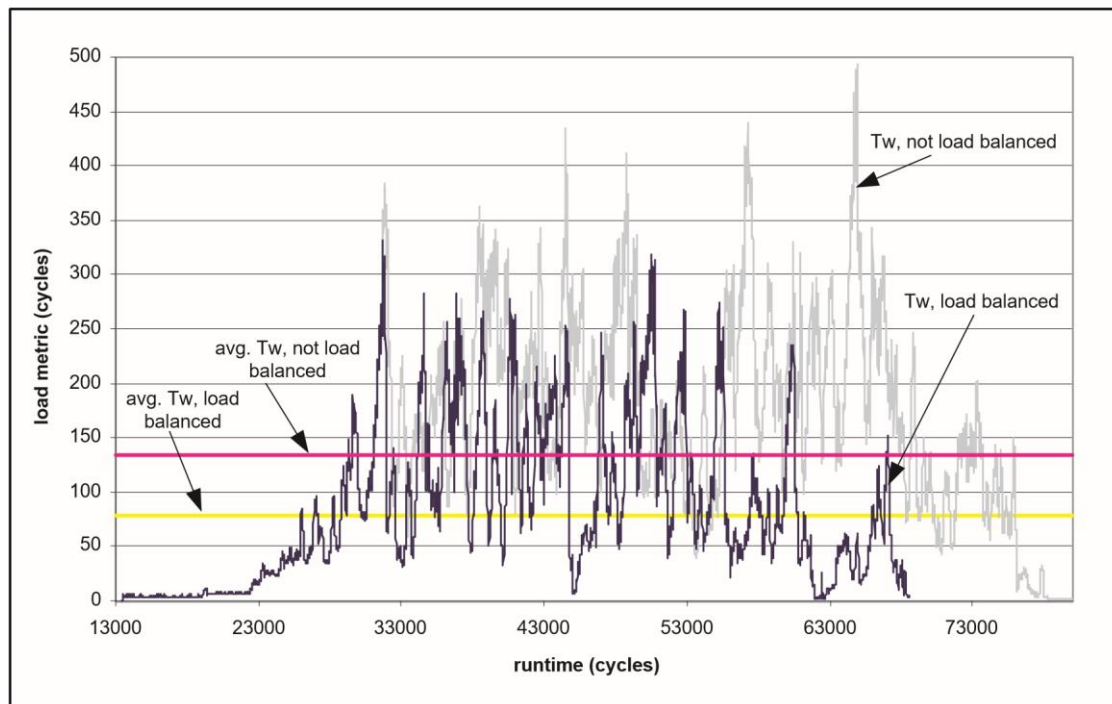


Рисунок 6-13: График зависимости метрики нагрузки от времени для теста быстрой сортировки с балансировкой нагрузки и без нее

Результаты механизма балансировки нагрузки в тесте Quicksort можно увидеть на рисунке 6-13. Этот рисунок показывает зависимость от времени для Quicksort из 200 элементов с балансировкой нагрузки и без нее с использованием номинальных метрик перехвата. Ускорение на 12,3% наблюдалось при использовании номинального порога перехвата; небольшое снижение порога перехвата и уменьшение интервала перехвата работы приводит к ускорению более чем на 15%. Однако я считаю, что эти более агрессивные пороги перехвата в целом не являются хорошей идеей; более частые пакеты обнаружения работы

перегружают сеть и будут иметь отрицательное влияние на производительность в приложениях с большим количеством межузловых коммуникаций.

Также была реализована выталкивающая функция потоков для исследования потенциальных преимуществ этого механизма. Выталкивающая функция потоков происходит только при создании нового потока. Такого рода выталкивающая функция потоков тривиальна для реализации на Q-Machine; это просто инструкция SPAWN, нацеленная на соседний узел. Опасность выталкивающей функции потоков заключается в том, что решение о выталкивании основано на устаревшей информации; кластер узлов может перегрузить один соседний незагруженный узел, если все загруженные узлы одновременно решили вытолкнуть работу на незагруженный узел. Несмотря на то, что опасность перегрузки невелика, поскольку в этом тесте быстрой сортировки есть только один исходный узел для потоков, выталкивающая функция потоков была реализована консервативно. Выталкивающая функция происходит только тогда, когда наблюдается метрика нагрузки соседа, близкая к нулю, а локальная метрика нагрузки очень высокая, более 400 циклов. В конечном итоге выталкивающая функция потоков использовалась редко и обеспечивала ускорение на 1–2 % в тесте быстрой сортировки.

Рисунок 6-14 более подробно иллюстрирует взаимосвязь между событиями миграции и . Из этого рисунка можно увидеть, как уменьшается с каждым событием миграции. На этом рисунке также показана нагрузка, возникающая на целевой объект миграции. Обратите внимание, что эта нагрузка сохраняется довольно низкой на протяжении всего бенчмарк-прогона.

Тест Quicksort демонстрирует, что реализация миграции Q-Machine достаточно эффективна, чтобы ускорить даже простой код, написанный без особых раздумий о параллелизме. Кроме того, механизм миграции достаточно быстр, чтобы обеспечить ускорение в тесте, который выполняется всего несколько десятков тысяч циклов. Напротив, большинству других механизмов миграции потребовалось бы в лучшем случае несколько тысяч циклов для завершения одной миграции нулевого потока.

### **6.3.2 Матрица МультиПликационное тестирование**

Для этого теста Бен Вандивером была написана пара ядер умножения матриц на языке People. Первое ядро использует один вложенный итеративный цикл для доступа к элементам матрицы и их умножения. Второе ядро использует потоки для умножения матриц. Потоки

являются уникальной особенностью языка People; по сути, они являются способом явного раскрытия базового

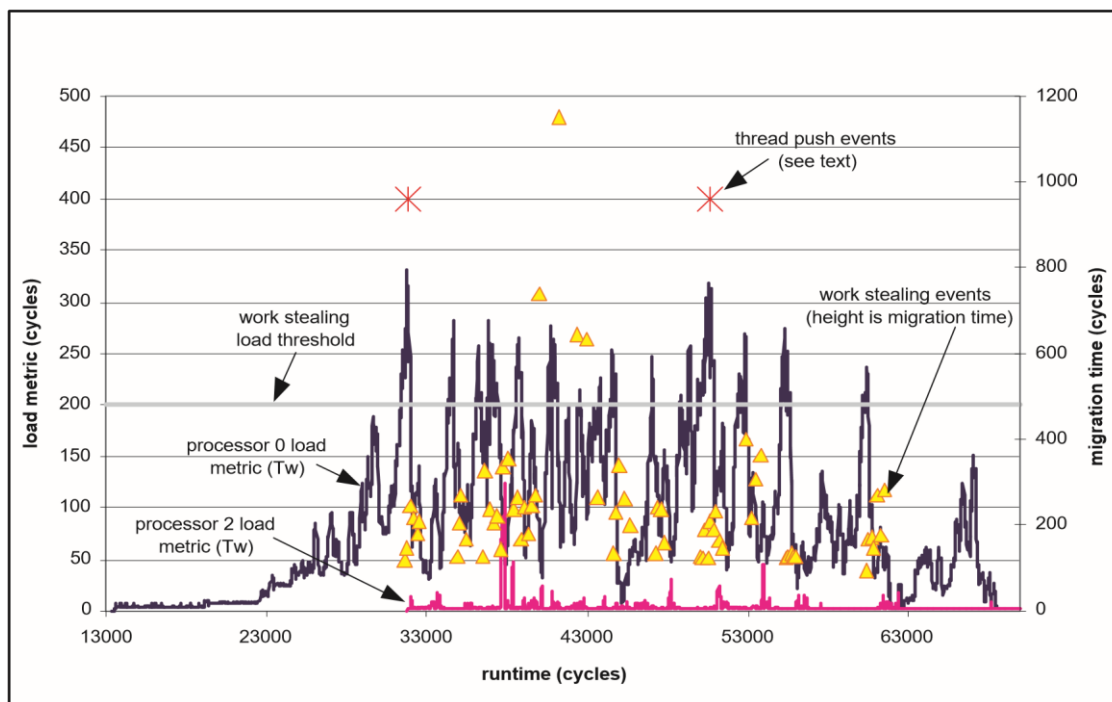


Рисунок 6-14: График результатов теста быстрой сортировки с балансировкой нагрузки и наложением событий миграции.

структуры очереди архитектуры для программиста. Ядро потокового матричного умножения создает два потока, которые являются источником данных матричного умножения, и потоковый оператор, который вычисляет и сохраняет результат умножения. Эти потоки позволяют выполнять вычисление индекса и доступ к массиву параллельно с фактической операцией умножения. Часть потокового кода матричного умножения показана на рисунке 6-

15. Поток называется модулем в People, его входы — источники, а выходы — стоки. Операции `pq()` и `dq()` используются для постановки в очередь и выписки из очереди данных в потоке соответственно.

В этом бенчмарке в качестве точки отсчета используется стандартное ядро умножения матриц; это чисто однопоточный фрагмент кода. С другой стороны, потоковое ядро умножения матриц создает три потока, по одному для источников матриц и один для



операции умножения. Таким образом, существует возможность для миграции данных сократить задержки доступа.

Я использовал очень простой алгоритм управления миграцией данных в этом бенчмарке. Каждые 200 циклов наиболее популярный элемент данных переносится в узел наиболее частого доступа. Наиболее популярный элемент данных определяется путем сохранения отсортированного, скользящего списка всех доступов в окне 4000 циклы.

Результат применения этого простого алгоритма миграции показан на рисунке 6-16 для размера 100x100.

```
модуль leftMat имеет sink[int], source[array[int]], source[int]
как "vals для левой стороны", "массив для использования",
"размер" внутренне source[int] vals, sink[array[int]] arr,
sink[int] s {

    int i,j,k; array[int]
    mat = dq(arr); int size
    = dq(s); i=0;
    пока (я < размер) {
        int смещение =
        i*размер; j=0;
        пока (j < размер) {
            k=1;
            nq(vals,mat[offset])
            ; пока (k < размер)
            {
                nq(vals,mat[k+offset]);
                к = к + 1;
            }
            j = j + 1;
        }
        я = я + 1;
    }
}

void matmult(размер int, массив[int] mat1, массив[int] mat2, массив[int] mat3) {
    приемник[целое] левая часть,
    правая часть;
    источник[массив[целое]] arr1,
    arr2; источник[целое] s1, s2;

    построить leftMat с lhs, arr1, s1;
    построить rightMat с rhs, arr2, s2;
    nq(arr1,mat1); nq(s1,size);
    nq(arr2,mat2); nq(s2,size);

    int i,j,k;
    i=0;
    пока (i < размер) {
        j=0;
        пока (j < размер) {
            k=0;
```

```

int сумма = 0;
пока (k < размер)
{
    сумма = сумма + dq(левая часть)*dq(правая часть);
    к = к + 1;
} mat3[j+i*size] =
сумма; print(сумма);
j = j + 1;
}
я = я + 1;
}
}

```

---

Рисунок 6-15: Часть теста умножения потоковой матрицы, написанного в People. умножение матриц, а на рисунке 6-17 — умножение матриц 15x15. Видно, что для умножения матриц 100x100 время на итерацию падает после первой итерации примерно с 7000 циклов до 1650 циклов на итерацию — ускорение примерно в 4,2 раза. Обратите внимание, что на рисунке 6-16 время первой итерации включает в себя накладные расходы на миграцию для перемещения двух матриц по 10 000 элементов.

При умножении матриц 15x15 миграция происходит позже, и время на итерацию увеличивается с 1100 циклов до примерно 350 циклов. Миграция происходит позже, поскольку наиболее популярные средства доступа — в данном случае источники потока умножения матриц — должны наращивать «популярность» по сравнению с потоком, который инициализировал матрицу через окно профилирования в 4000 циклов. Это приводит к ускорению в 3,2 раза.

Также интересно отметить, что потоковые реализации превосходят однопоточную реализацию умножения матриц примерно в два раза в каждом случае бенчмарка. Это положительный показатель преимуществ производительности потоковых функций в языке People. В этом конкретном случае ускорение является результатом распараллеливания (развязки, для любителей архитектур DAE) доступа к памяти и операции умножения.

Ускорения на итерацию в потоковых тестах полностью обусловлены сокращением задержки, вызванным миграцией данных; балансировка нагрузки не влияет на результаты, поскольку тест использует только три потока. Из результатов теста можно увидеть, что многопоточные потоковые реализации без миграции данных на самом деле работают хуже, чем однопоточная реализация. Это связано с тем, что People не учитывает размещение памяти относительно потоковых потоков, поэтому хорошая производительность зависит от

наличия специализированного механизма миграции, который уменьшает задержку. В этом конкретном случае тест выполнялся на машине с 16 узлами, и исходные данные для каждого из потоков были перемещены на расстояние двух переходов маршрутизатора каждый. Этот шаг снижает задержку доступа в лучшем случае с 55 циклов до 7 циклов. Обратите внимание, что другой подход к решению этой проблемы задержки доступа заключается в использовании лучших методов сокрытия задержки в коде и в том, чтобы исходные потоки не ждали возвращения каждой загрузки из узла памяти перед пересылкой данных на потоковый множитель. Однако это проблема компилятора, и одним из основных моментов этого бенчмарка является демонстрация того, что миграция данных может быть успешно применена к пользовательской программе.

### 6.3.3 N-Body бенчмарк

Для этого бенчмарка я написал гравитационный симулятор N-Body в People. Используемый алгоритм — это базовый метод частица-частица, где каждое тело вычисляет чистый вклад силы каждого другого тела на каждом шаге времени. Для решения разностной задачи использовался метод Рунге-Кутты второго порядка.

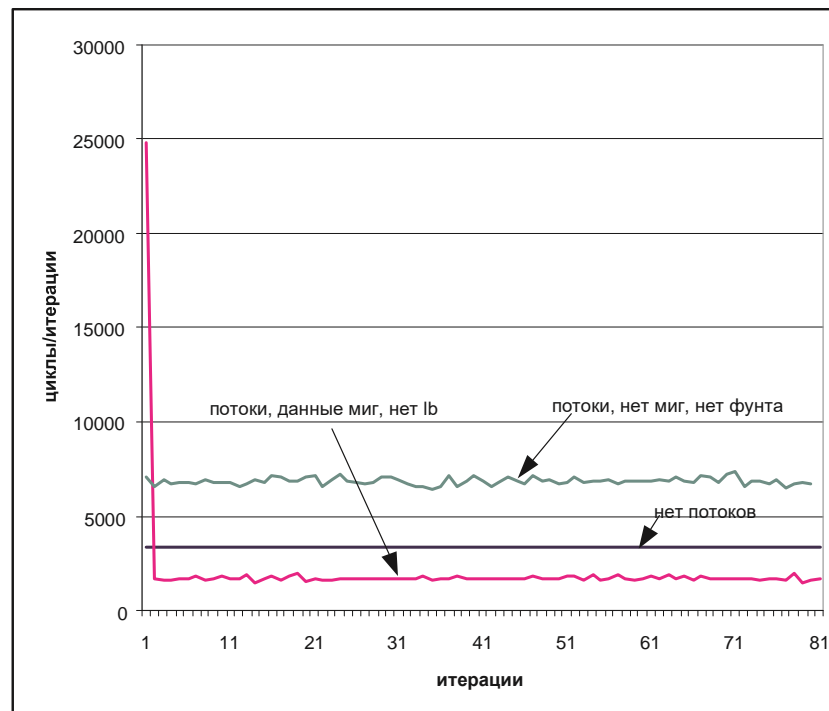


Рисунок 6-16: График времени, необходимого для итерации умножения матрицы 100x100 при различных условиях миграции и стилях кодирования.

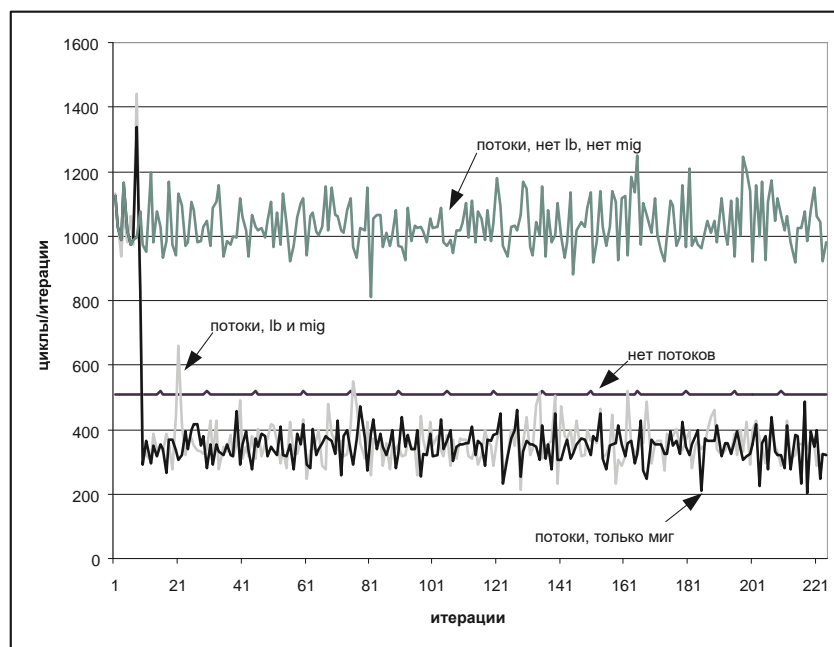


Рисунок 6-17: График времени, необходимого для итерации умножения матрицы 15x15 при различных условиях миграции и стилях кодирования. Уравнение силы в основе метода частица-частица. Числовое ядро этого кода взято из [Che], [Sch] и [Har00]. Графическое представление выходных данных гравитационного моделирования N-Body можно увидеть на рисунке 6-18. На этом рисунке показаны первые несколько временных шагов моделирования 12 тел, запущенного на 64-узловом Q-Machine.

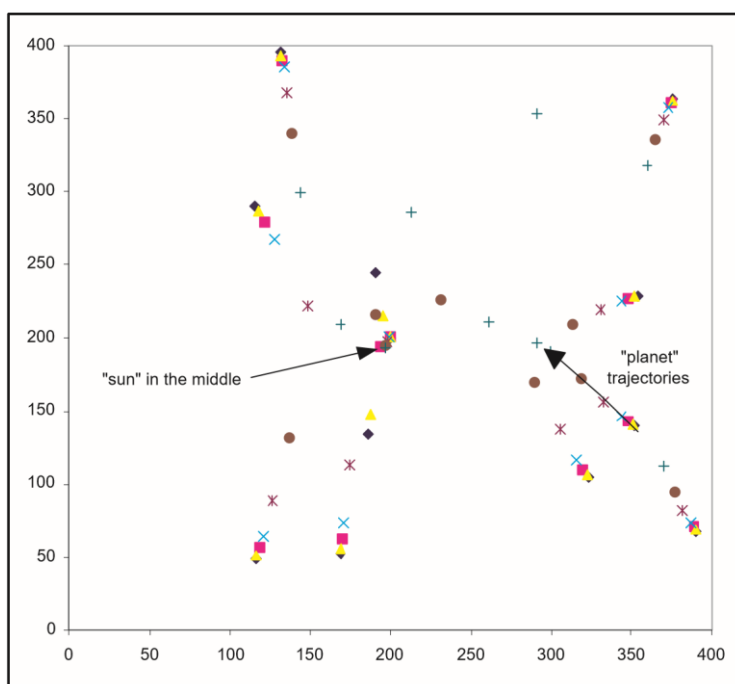


Рисунок 6-18: График первых нескольких временных шагов выходного сигнала теста N-Body

Моя первоначальная реализация N-Body создавала поток на планету и использовала двоичное дерево объектно-ориентированных семафоров для определения завершения каждой итерации; Бен Вандивер оптимизировал это, чтобы использовать дерево, состоящее из потоков, для сигнализации о завершении каждой итерации. Поточковые оптимизации Бена и несколько других настроек сократили время на итерацию бенчмарка N-Body примерно в 4 раза сами по себе. Оптимизации Бена включали статическое размещение данных и оптимизацию повторного использования, поэтому мало что можно выиграть от динамической миграции данных; единственное, что имеет значение, это то, что начальные данные распределяются примерно равномерно по всем узлам машины. Тщательное начальное размещение данных важно, поскольку фактическое вычисление силы гравитации выполняется методом объекта, вызываемым деревом объединения, а вызовы методов объекта всегда появляются рядом с экземпляром объекта переменные.

Несмотря на оптимизации, ускорение на 36% было достигнуто путем применения программно-управляемой миграции с задержкой к бенчмарку N-Body. Этот результат можно увидеть в циклах на время итерации, представленных на рисунке 6-20. Чтобы понять, как миграция потоков с задержкой использовалась для ускорения бенчмарка N-Body, нужно сначала понять структуру кода бенчмарка.

Внутренний цикл бенчмарка N-Body состоит из цикла, который посещает каждую из планет и вычисляет их частичный вклад силы на локальной планете. См. рисунок 6-19. Перед входом во внутренний цикл все переменные экземпляра объекта локальной планеты извлекаются во временные объекты, которые компилятор хранит в очередях. Эти переменные экземпляра записываются обратно в объект планеты при выходе из внутреннего цикла. В начале каждой итерации цикла удаленная планета выбирается оператором `Body body = planets[jj];`. Затем внутренний цикл вычисляет вклад силы удаленной планеты; во время этого вычисления цикл многократно ссылается на переменные экземпляра удаленной планеты. Поскольку удаленная планета обычно находится на другом узле, эти ссылки на память довольно медленные без какого-либо механизма для сокращения задержки доступа. Поэтому в строке, следующей сразу за инициализацией тела, я вызываю системную функцию `migrate(Body)`. Эта функция заставляет локальный поток немедленно отменить планирование

и перенести себя на домашний узел аргумента. Этот вызов `migrate()` уменьшает время вычисления внутреннего цикла на 36%. Это ускорение полностью обусловлено сокращением задержки доступа к переменным экземпляра удаленной планеты. Следовательно, ускорение пропорционально количеству ссылок на переменные экземпляры удаленного объекта во внутреннем цикле. Например, упрощение решателя дифференциальных уравнений N-Body для использования менее точного, но более быстрого метода Эйлера приводит к тому, что миграция потоков становится неэффективной, поскольку в методе Эйлера требуется всего три или четыре доступа к переменным экземпляра. Напротив, метод Рунге-Кутты второго порядка, используемый для получения этих результатов, требует девятнадцати доступов к переменным экземпляра. Обратите внимание, что миграция данных никогда не является эффективным методом ускорения приложения N-Body в том виде, в котором оно написано, поскольку в любой момент времени несколько потоков, расположенных на нескольких узлах, обращаются к переменным экземпляра объекта.

```
float ax=this.ax; // загрузка локальных переменных планет в
очереди float ay=this.ay; // и т.д...

int jj = 0; пока (
jj < размер ) {
    если((myIndex != jj)) {
        Тело body = planets[jj]; // доступ к удаленной планете
        migrate(body);          // перенести 'это' на узел планеты

        rad = (x-body.x)*(x-body.x) + (y-body.y)*(y-body.y);

        // Ядро Рунге-Кутты опущено для ясности...

        топор = (ax1 +
топор2)/2,0; ай = (ай1 +
ай2)/2,0;
    }
    jj = jj + 1;
}

this.ax = ax; // записать обратно локальные переменные
планеты this.ay = ay; // и т. д...
```

---

Рисунок 6-19: Внутренний цикл кода эталонного теста N-Body.

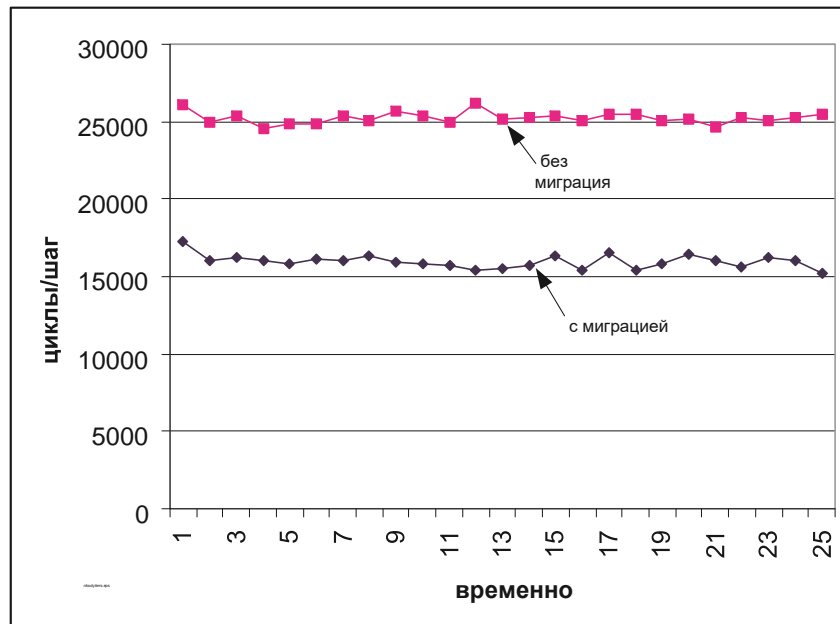


Рисунок 6-20: График времени, необходимого для каждого временного шага моделирования 12 тел N-тел, запущенного на 64-узловой Q-машине.

## Глава 7

# Выводы и будущая работа

...но мне нравятся большие ключи. Кого волнует, что нет болта достаточно большого для этого ключа? Кто-нибудь когда-нибудь его сделает.

— *Доминик Риццо*

### 7.1 Выводы

В этой диссертации описывается и демонстрируется абстрактная архитектура машины, ADAM, которая обеспечивает высокопроизводительные механизмы миграции в оборудовании. Архитектура ADAM включает в себя вездесущие очереди, которые служат гибкой, единообразной аппаратной абстракцией для обмена потоками и памятью. Эти очереди также служат для разделения таймингов потоков и памяти. Архитектура ADAM также включает в себя систему памяти на основе возможностей, которая обеспечивает быстрое разрешение границ данных и принудительное выполнение проверок границ в оборудовании. Наконец, архитектура включает в себя модель программирования с массивной многопоточностью, где потоки являются просто частными случаями возможностей данных, так что механизм, аналогичный используемому для миграции данных, может быть применен и к потокам. Массовая многопоточная природа архитектуры также позволяет скрыть задержку путем переключения контекста при событиях остановки потока.

Особенности архитектуры ADAM объединяются для обеспечения эффективного и быстрого механизма миграции данных и потоков. Этот механизм миграции опирается на два протокола, реализованных в оборудовании: удаленный поиск данных с помощью указателей локатора данных и временные двунаправленные указатели для обновления указателей после



событий миграции. Сам механизм миграции довольно прост; по сути, алгоритм — «заморозить, скопировать и переслать».

Производительность моего механизма миграции была продемонстрирована в нескольких тестах. Эти тесты были запущены на преимущественно точной по циклам машинной симуляции, написанной на Java. Тесты были выбраны для демонстрации как возможностей балансировки нагрузки, так и возможностей сокращения задержек реализации. Тест быстрой сортировки на месте показал увеличение производительности на 12,3% с простыми алгоритмами балансировки нагрузки с захватом работы и выталкиванием потоков. Тест потокового матричного умножения показал увеличение производительности в 4 раза при применении миграции данных с учетом задержек. Наконец, тест моделирования гравитации N-Body продемонстрировал ускорение на 36% при применении миграции потоков с учетом задержек.

Мой механизм миграции работает на порядки быстрее, чем предыдущая работа. Active Messages [WGQH98], высокопроизводительная программная реализация миграции потоков, может push-мигрировать 6000 нулевых потоков в секунду; это соответствует примерно 300 с на событие миграции или около 16 000 циклов процессора при тактовой частоте процессора 50 МГц, указанной в статье. Моя архитектура может push-мигрировать нулевые потоки примерно за 70 циклов, что соответствует примерно 70 нс на событие миграции в предлагаемой аппаратной реализации. Это соответствует ускорению примерно в 5000 раз. Производительность моего механизма миграции достаточно высока, чтобы быть привлекательной даже по сравнению с традиционными механизмами сокрытия задержек, такими как кэши: заполнение кэша L2 Pentium 4 занимает около 175 нс или 140 циклов RAMBUS [CJDM01]. Надеюсь, производительность схемы миграции моей диссертации сделает ее вариантом для управления задержками в будущих реализациях высокопроизводительных параллельных компьютеров.

## 7.2 Будущая работа

Есть много интересных направлений для изучения в будущей работе. Наиболее важным вопросом для рассмотрения будут алгоритмы для эффективного управления механизмом миграции. Другими вопросами, требующими внимания, будут языки программирования и

среды разработки для архитектуры. Наконец, архитектура должна быть приведена к практике с реальной аппаратной реализацией.

### 7.2.1 Улучшенные алгоритмы управления миграцией

В моей работе описаны некоторые простые метрики и алгоритмы для управления механизмом миграции. Хотя эти алгоритмы показали себя достаточно хорошо, чтобы показать здоровый рост производительности на нескольких бенчмарках, они далеки от оптимума или полноты. Мой опыт работы с моими простыми алгоритмами управления миграцией показывает, что эти вопросы будут важны при внедрении и тестировании будущих алгоритмов управления:

**Метрики.** Метрики необходимы для суммирования задержки связи и загрузки процессора в алгоритме управления. Понимание этих метрик и выбор правильных данных для отчета являются предпосылкой для принятия разумных решений по управлению. Хорошая метрика также должна быть независимой от реализации и масштабируемой с помощью технологий.

**Гибкость.** Хотя можно показать, что алгоритм управления является оптимальным при наборе ограниченных условий эксплуатации, полезный алгоритм управления должен иметь ограниченную производительность в широком диапазоне условий эксплуатации.

**Прочность.** Аппаратные сбои неизбежны в любой крупной системе. Хороший алгоритм управления должен быть устойчив к аппаратным сбоям; было бы предпочтительнее, если бы алгоритм был достаточно умен, чтобы перемещать данные из неисправных узлов.

**Повторные попытки.** В реализации Q-Machine заблокированные сообщения отправляются повторно; каждая попытка доставки увеличивает эффективную задержку сообщения на величину, пропорциональную времени задержки повторной отправки. Интеллектуальный алгоритм управления миграцией должен распознавать такие ситуации и прерывать процесс миграции, если дополнительная задержка повторной отправки пакета миграции повредит общей производительности.

### 7.2.2 Язык и компиляторы

ADAM разрабатывался параллельно с языками, которые могли использовать его уникальные возможности; фактически, в ходе разработки Бен Вандивером [Van02] были разработаны два языка: Couatl и People.

Couatl — первый язык, разработанный для платформы ADAM; это простой объектно-ориентированный язык, который использует технику, известную как постоянные методы, для выполнения диспетчеризации объектов. Постоянные методы запускаются один раз для каждого экземпляра объекта и никогда не завершаются. Каждый объект имеет связанный с ним постоянный метод, который действует как сервер для вызовов методов. Постоянный серверный метод ожидает, пока клиент поставит запрос на вызов метода в назначенную очередь запросов. Получив запрос, серверный метод выполняет поиск метода и порождает новый поток выполнения для этого метода. Couatl был в первую очередь разработан для доказательства того, что модель программирования ADAM жизнеспособна и что разумный анализ компилятора может генерировать код, который использует очереди без взаимоблокировки. Он также доказал, что модель «поток на метод» является жизнеспособной моделью программирования. Основные проблемы с Couatl включают отсутствие видимости структур очередей на уровне программиста и отсутствие встроенной поддержки пространственной осведомленности. Код, сгенерированный с помощью Couatl, размещал все методы и новые объекты на одном узле и использовал механизм балансировки нагрузки для повышения производительности.

People (от PPL, Parallel Programming Language) является преемником Couatl. People также использует объектно-ориентированную модель программирования. Его наиболее существенным дополнением является поддержка потоковых конструкций, которые открывают программисту очереди в ядре машины. Потоки представляют собой способ для программиста явно планировать статические шаблоны связи. Эти потоковые конструкции использовались в тесте Matrix Multiply, например, для настройки статического конвейера доступа к массиву/умножения. Они также использовались в тесте N-Body для создания статического дерева объединения для определения того, когда все потоки завершили вычисление своего результата на каждом временном шаге.

Будущие языки для архитектуры ADAM должны также включать примитивы для полос, расщепления и рассеивания массивов и векторов. Также был бы полезен оператор

параллельного отображения, а также механизмы для упрощения построения деревьев разветвления и разветвления. Также требуется поддержка сборки мусора во время выполнения. Для эффективной реализации параллельной сборки мусора обратитесь к Jeremy Диссертация Брауна о редкогранных массивах (SFA) и масштабируемой параллельной сборке мусора. [Bro02]

### **7.2.3 Аппаратная реализация**

Важным шагом в развитии любой архитектуры является ее аппаратная реализация. К счастью, сама природа абстрактной машинной архитектуры поддается инкрементальной реализации. Кроме того, сеть fat-tree реализации Q-Machine dilation-2 имеет встроенную отказоустойчивость; [DeH93] подробно описывает, как сетевая реализация может выдержать единичный сбой в любом компоненте или соединении без потери логической связности. Наконец, архитектура ADAM и реализация Q-Machine были разработаны с учетом практических вопросов выхода годных и сопротивления устареванию. Выход годных на кристалле многопроцессорного чипа Q-Machine может быть близок к 100%. Благодаря аппаратной абстракции ADAM чипы с несколькими неисправными процессорами по-прежнему пригодны для продажи; системе выполнения просто нужна карта неисправных мест, чтобы никакие данные или потоки не переносились в сломанные узлы. Аппаратная абстракция ADAM также помогает продлить срок службы архитектуры; по мере выхода узлов из строя их можно заменить новыми узлами, реализованными в новейшей технологии процесса, без необходимости перекомпиляции. В сочетании с системой управления миграцией, которая может перемещать данные и потоки из неисправного узла, система может работать годами, постоянно обновляясь, и при этом не останавливаться.

### **7.2.4 Транзакции**

Аппаратная поддержка транзакций очень полезна в больших параллельных архитектурах. Откат транзакций обеспечивает более высокий уровень спекулятивного параллелизма, а транзакционные контрольные точки обеспечивают более высокий уровень динамической отказоустойчивости. Архитектура ADAM имеет некоторые уникальные особенности, которые позволяют в будущем реализовывать транзакции на оборудовании.

Первое наблюдение заключается в том, что очередь может быть превращена в транзакционный журнал, если операция «dequeue» обратима. Другими словами, оператор

dequeue должен просто продвигать указатель dequeue, не отбрасывая никаких данных. Учитывая это преобразование, вычислительное состояние потока ADAM можно сохранить, просто запомнив все смещения указателей enqueue и dequeue. Чтобы вернуть память, вычисление может быть зафиксировано после того, как необходимые условия выполнены, отбросив некоторые данные. Состояние памяти также можно сохранить с помощью этой схемы, используя только операторы обмена в памяти и обращения обмена во время отката.

Второе наблюдение заключается в том, что состояние потока ADAM полностью представлено в пределах возможностей потока. Следовательно, контрольная точка может быть выполнена на более грубой зернистости, чем ранее предложенный метод отката указателя, просто путем создания копий состояния потока. Этот механизм грубой зернистости транзакций проще в реализации и требует меньше аппаратных модификаций.

Конечно, дьявол кроется в деталях. Многие проблемы, такие как то, как справиться с миграцией и недетерминированным порядком сообщений между потоками, должны быть решены, прежде чем любая из схем может быть объявлена успешной.

## **7.3 Заключительные замечания**

Как указано в разделе «Будущая работа», архитектура ADAM и реализация Q-Machine полны легких путей. Кроме того, архитектура ADAM имеет последствия для высокопроизводительных параллельных компьютеров, выходящие за рамки простого обеспечения высокопроизводительной схемы миграции данных и потоков. Я надеюсь изучить эти возможности в ближайшем будущем. Я также призываю всех, кто нашел время прочитать мою диссертацию, изучить и реализовать аспекты архитектуры, и, пожалуйста, не стесняйтесь отправлять мне электронные письма, если у них есть какие-либо вопросы. Мой пожизненный адрес электронной почты — [bunnie@alum.mit.edu](mailto:bunnie@alum.mit.edu). Я с нетерпением жду вашего ответа.

## Приложение

А

## Сокращения

Я

В  
:  
К  
а  
к  
в  
ы  
д  
у  
м  
а  
е  
т  
е  
,  
ч  
т  
о  
б  
у  
д  
е  
т  
с  
а  
м  
о  
й  
б  
о  
л  
ь

ш  
о  
й  
п  
р  
о  
б  
л  
е  
м  
о  
й  
в  
в  
ы  
ч  
и  
с  
л  
и  
т  
е  
л  
ь  
н  
о  
й  
т  
е  
х  
н  
и  
к  
е  
в  
9  
0  
-  
х  
?  
О  
:  
С  
у  
щ  
е

с  
т  
в  
у  
е  
т  
в  
с  
е  
г  
о  
1  
7  
0  
0  
0  
т  
р  
е  
х  
б  
у  
к  
в  
е  
н  
н  
ы  
х  
а  
б  
б  
р  
е  
в  
и  
а  
т  
у  
р  
.

—

*Поль*

*Бутен*



изНов  
ый  
словар  
ь  
хакера

В этой главе для удобства  
читателя перечислены  
сокращения и  
аббревиатуры,  
используемые в данной  
диссертации.

\$	Сокращение для кэша
АСК	Сокращение для подтверждения
АДАМ	Овен Децентрализованная Абстрактная Машина
ЯВЛЯЮСЬ	Память об аттракционе
АМД	Современные микроустройства
ASIC	Специализированная интегральная схема
ЖОПА	Симулятор системы ADAM
БИСТ	Встроенная самопроверка
ccNUMA	Неравномерный доступ к кэш-когерентной памяти
СМ	Машина для соединения
КМОП	Комплементарный металл-оксид-полупроводник
КМП	Многопроцессорная или химико-механическая полировка чипа
КОМА	Архитектура кэш-памяти
Коатль	Производный от Java объектно-ориентированный протоязык для ADAM
Процессор	Центральный процессор
Технический директор	Главный технический директор
ДАЕ	Разделенный доступ Выполнение
ДДМ	Машина распространения данных
ДМА	Прямой доступ к памяти
ДМЭМ	Память данных
DRAM	Динамическая оперативная память
ЕСС	Код исправления ошибок
ЭМЭМ	Кэш среды
ИСПОЛНИТЕЛЬНЫЙ	Исполнительный блок ядра Q-Machine
ФО	Развернуть
I\$	Кэш инструкций, часто неправильно используемый для обозначения очереди планировщика гибридного рабочего окна.
ИБМ	Международная Бизнес Машина
ИДЕНТИФИКАТОР	Сокращение для идентификатора
ИС	Указатель инструкций или интеллектуальная собственность
МПК	Инструкции на такт
ИСА	Архитектура набора инструкций
КСР	Исследование площади Кендалл
МЗП	Наименее значимый бит
МИМД	Множественные инструкции, множественные данные

Массачусетский технологический институт	Массачусетский технологический институт
МСБ	Самый значимый бит
НИ	Сетевой интерфейс
СЕЙЧАС	Сеть рабочих станций
НСФР	Поименованный государственный регистрационный файл
НУМА	Неравномерный доступ к памяти

Таблица А.1: Таблица сокращений

120

ОРБ	Брокер запросов объектов
ОСИ-7	Семиуровневая модель взаимодействия открытых систем
ПК	Счетчик программ или персональный компьютер
Люди	Язык второго поколения для ADAM, поддерживающий потоковые конструкции
ФИЗИЧЕСКИЙ	Физический и канальный уровни (из модели OSI-7)
ПИМ	Процессор в памяти
ЗПП	Язык параллельного программирования (он же People)
Пкф	Файл физической очереди
Q\$ или QC	Кэш очереди
РПК	Удаленный вызов процедуры
РИСК	Компьютер с сокращенным набором команд
РАСПИСАНИЕ	Планировщик Сопроцессор
СФА	Редкограненый массив
СГИ	Silicon Graphics Incorporated
СИА	Ассоциация полупроводниковой промышленности
СМЭМ	Планировщик Сопроцессор Память
СМТ	Технология одновременной многопоточности или поверхностного монтажа
СОИ	Кремний на изоляторе
SRAM	Статическая оперативная память
SSRAM	Синхронная статическая оперативная память (SRAM)
источник	аббревиатура для Источника

131

TAM	Поточная абстрактная машина
TSMC	Тайваньская корпорация по производству полупроводников
TTDA	Архитектура потока данных с тегированными токенами
VLIW	Очень длинное слово-инструкция
ВКФ	Файл виртуальной очереди
ВКН	Номер виртуальной очереди
XPRT	Сетевые и транспортные уровни (из модели OSI-7)

Таблица А.2: Таблица  
сокращений, продолжение  
122

# Приложение Б

## Подробнос ти АДАМ

Мои два  
любимых  
языка — это  
по-  
прежнему  
ассемблер и  
пайка.

—  
*кролик*

В этом приложении  
приводятся многие детали,  
опущенные в главе 3 в целях  
ограничения основного  
текста диссертации только  
функциями и вопросами,

относящимися к миграции. Обратите внимание, что в этом приложении не обосновываются все решения по проектированию так же строго, как в основных главах диссертации; на самом деле, некоторые решения по реализации, такие как использование упрощенного формата с плавающей точкой, в основном являются результатом моих собственных предубеждений. С другой стороны, выбор формата с плавающей точкой имеет мало общего с сутью архитектуры, и я представляю такой материал здесь только потому, что могу.

## **Б.1 Типы данных**

Все типы данных ADAM имеют ширину 80 бит; они состоят из 64-битного поля данных и 16-битного поля

тега. Поддерживаются  
четыре целочисленных типа  
данных: знаковое длинное  
(называемое «словом»),  
упакованное знаковое  
целое, упакованное  
знаковое короткое и  
упакованные символы  
юникода. Поддерживается  
только один тип данных с  
плавающей точкой,  
аналогичный формату  
IEEE-754 double.

Подробное форматирование  
типов данных на уровне бит  
см. на рисунке В-1.

Упакованные данные  
обрабатываются в  
векторной форме;  
большинство  
арифметических операций  
поддерживаются на  
упакованных данных.  
Однако любая  
арифметическая операция,  
включающая возможность,  
действительна только для  
слова. Однако для  
смещений очереди памяти  
поддерживается любой  
целочисленный тип. Более  
подробную информацию о

модели памяти ADAM см. в разделе 3.2.3.

Все типы данных полностью помечены для идентификации их типа, а также любых флагов, связанных с их статусом. Подробности см. на рисунке В-2. Ошибки в арифметических операциях могут быть принудительно перехвачены и

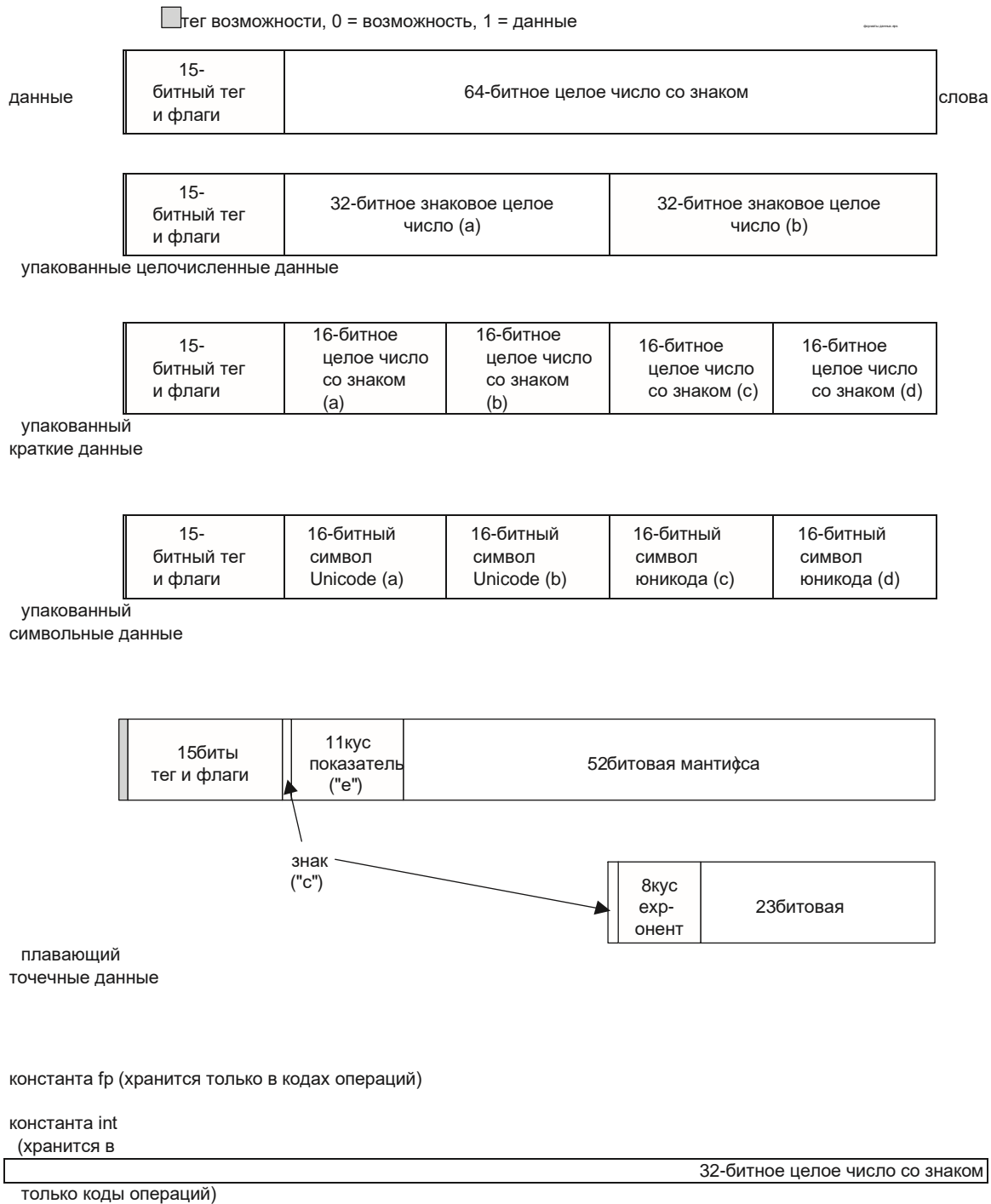


Рисунок В-1: Форматы данных, поддерживаемые ADAM не-перехватывающие. Ошибки перехвата приводят к остановке потока и выдаче исключения; ошибки неперехватывающие позволяют нормальному выполнению (что может подразумевать или не подразумевать остановку), а состояние ошибки просто указывается в поле тега и флагов результата. Это состояние ошибки будет распространяться через



операции с данными; другими словами, добавление float с тегом NaN к допустимому float приведет к float с тегом NaN.

Неизменяемый бит включен в теги для указания статических данных, которые не могут быть изменены. Идентификация данных как статических позволяет процедурам управления свободно копировать неизменяемые данные, тем самым обеспечивая дешевые автоматические механизмы для распространения часто используемых констант. Запись в данные, объявленные как неизменяемые, не оказывает влияния на данные, может вызвать исключение и всегда устанавливает бит в регистре состояния, чтобы указать, что произошла недопустимая запись.

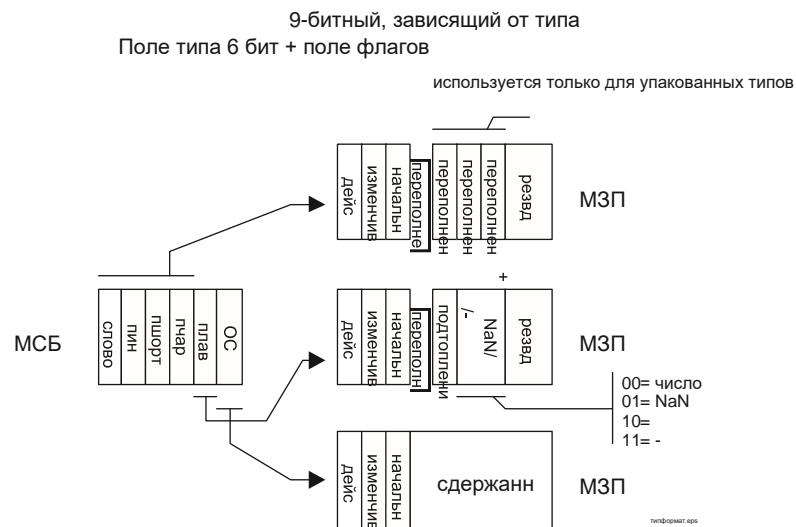


Рисунок В-2: Подробности полей тегов и флагов

В каждый тег также включен основной бит, который используется менеджером миграции данных для указания, является ли это основной копией данных. Это особенно полезно для сценария частичной миграции, когда основная возможность, содержащая некоторые данные, была перенесена, но сами данные еще не перемещены. Подробнее о механизмах миграции и ее реализации см. в главе 4.

Подмножество стандарта IEEE 754-1985 с плавающей точкой требуется архитектурой ADAM. Различия между стандартом IEEE 754-1985 и форматом ADAM выбраны для упрощения реализации и повышения производительности с небольшим снижением точности. Эти различия

являются:

ADAM не поддерживает числа с плавающей точкой одинарной точности и связанные с ними операции и преобразования, за исключением константных полей в кодах операций. NaN и указаны в поле тега, поэтому теперь допустима экспонента = 2047, а смещение экспоненты теперь равно +1024

ADAM не имеет денормализаций (точность по сравнению с IEEE 754-1985 восстанавливается путем указания специальных типов чисел в поле тега, как описано выше) один режим округления: округление в стиле фон Неймана

Подводя итог, значение числа с плавающей точкой равно , если только и , в этом случае значение равно (ноль со знаком).

Помимо этих различий, формат с плавающей точкой ADAM соответствует стандарту IEEE 754-1985 [Ste85]. В частности, обработка NaN, бесконечности и знакового нуля в контексте исключений, ловушек, сравнений и преобразований идентична.

Формат инструкций ADAM позволяет хранить 32-битные константы в стандартном коде операции. Таким образом, инструкции с плавающей точкой могут хранить число с плавающей точкой одинарной точности в поле константы, но оно немедленно преобразуется в число с двойной точностью при использовании. Аналогично числа с плавающей точкой одинарной точности не имеют представления деноминации; следовательно, NaN и не могут быть представлены в поле констант с плавающей точкой одинарной точности. Значение числа с плавающей точкой одинарной точности является пока не и , в этом случае значение равно (подпись) ноль).

Округление в стиле фон Неймана реализуется путем добавления младшего значащего бита (LSB) точности к числам с плавающей точкой, когда числа с плавающей точкой поступают в арифметический конвейер, и переноса этого младшего значащего бита точности по всему конвейеру. Этот дополнительный LSB устанавливается в двоичную «1», когда числа поступают в конвейер, и округление выполняется путем простого усечения в конце конвейера. Это приводит к тому, что ожидаемое значение дополнительного LSB будет \_в конце дня.


Реализация может выбрать использование полного округления в стиле IEEE 754-1985 для получения дополнительной точности, но в спецификации стандартной архитектуры нет

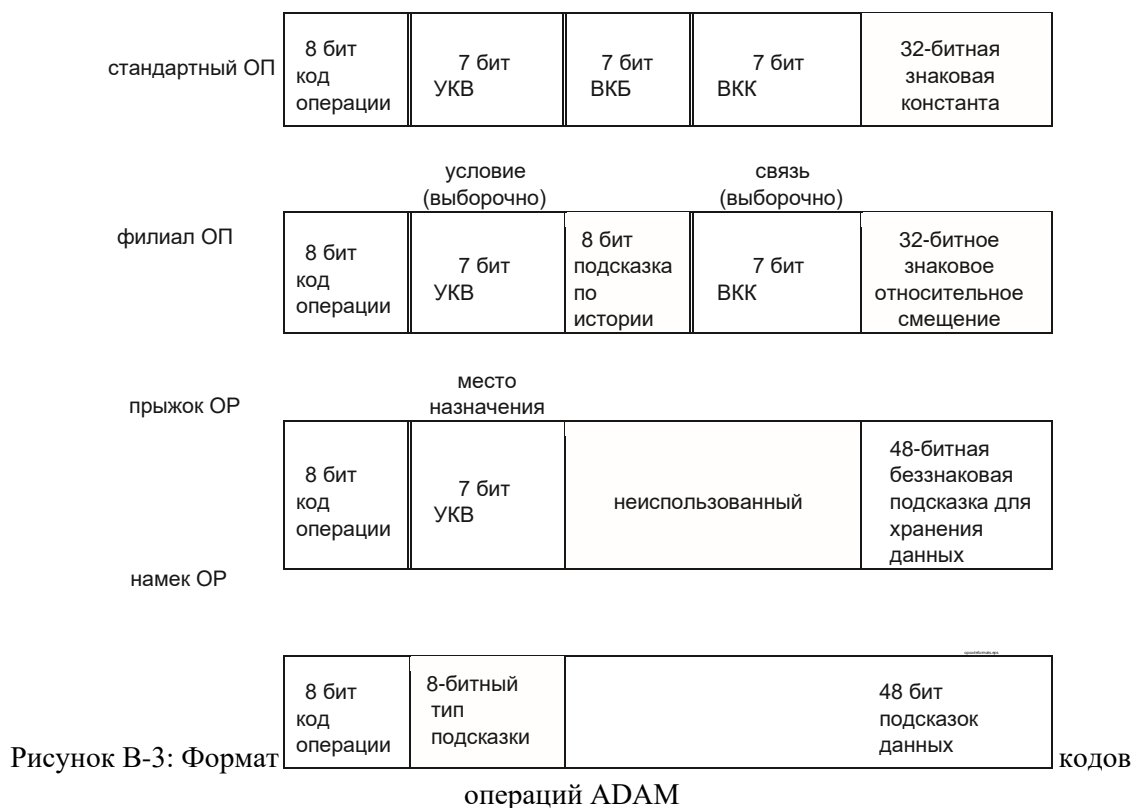
положения, позволяющего выбрать, какой режим округления использовать; таким образом, режимом округления по умолчанию и единственным режимом округления должно быть «округление до ближайшего» согласно IEEE 754:1985.

## Б.2 Форматы инструкций

ADAM имеет изолированное кодовое пространство, как в архитектуре Гарварда. Кодовое пространство, в отличие от пространств данных и окружения, является глобальным и общим для всех узлов; это осуществимо, поскольку кодовое пространство в основном доступно только для чтения. Управляющий сопроцессор заботится об обработке любых ошибок страниц или загрузке и выгрузке кода в кодовом пространстве. ADAM может динамически запрашивать новые классы объектов для загрузки в кодовое пространство с помощью инструкции LDCODE.

Пространство кода в основном доступно только для чтения, поскольку некоторые инструкции содержат поля подсказок для предварительной выборки инструкций. Фактические значения, содержащиеся в полях подсказок, зависят от реализации, и любая реализация ADAM должна выполнять код правильно независимо от содержимого поля подсказок; однако компилятор может свободно разогревать поля подсказок с помощью битовых шаблонов, которые могут улучшить производительность запуска для конкретной реализации. Кэши инструкций могут заменять строки, которые не были записаны обратно из-за нехватки пропускной способности памяти инструкций, без какого-либо влияния на правильность выполнения. Аналогично, значения записи не должны распространяться по всей системе, даже если код глобально распределяется между всеми узлами. Однако в случае, если значения возвращаются в исходный файл на диске, при следующей загрузке кода он может работать быстрее.

тег копирования/замены, 1 = копирование/замена, 0 = исключение из очереди/постановка в очередь



Инструкции имеют длину 64 бита и четыре основных формата: стандартный, ветвь, переход и подсказка (см. В-3). Каждая инструкция имеет 8-битное поле кода операции. Каждый спецификатор очереди в каждой инструкции изменяется битом копирования/затирания. Установка тега копирования/затирания позволяет компилятору обрабатывать очередь с семантикой, похожей на семантику регистра. Операция копирования извлекает значение из очереди, не изменяя никаких значений в очереди; операция затирания проверяет, пуста ли очередь, и если это так, ждет, пока в нее не будет записано значение, а затем заменяет значение. Операция затирания недопустима в переотображенной очереди, и попытка выполнить такую операцию вызывает исключение.

Стандартная инструкция имеет три спецификатора виртуальной очереди, каждый длиной 7 бит. Первые два (VQA и VQB) определяют очереди чтения; последний (VQC) определяет очередь записи. Стандартная инструкция также содержит 32-битное знаковое константное поле, что позволяет стандартной инструкции указывать до трех источников данных и одно место назначения данных, хотя большинство инструкций не используют это преимущество.

Определенные инструкции, известные как инструкции специального формата, могут интерпретировать VQA, VQB или

Поля VQC как константы, а не как очередь для ссылки на извлечение или сохранение данных в файле очереди. Эти инструкции обычно касаются создания, обслуживания и уничтожения карт очередей. Компилятор и/или программист на языке ассемблера обычно всегда знает точный номер очереди, к которой применяется отображение, поэтому для большинства инструкций по обслуживанию карт очередей не имеет смысла принимать произвольные динамически сгенерированные значения очереди. Следовательно, поля VQA, VQB и VQC можно использовать для немедленного обращения к номеру очереди для этих инструкций.

Инструкции ветвления имеют поле условия, поле ссылки, поле подсказки истории ветвления и 32-битное смещение ветвления со знаком. В инструкции можно опустить либо поле условия, либо поле ссылки, но не оба сразу. Также предусмотрено 8-битное поле подсказки истории, чтобы можно было сохранить историю ветвления вместе с инструкцией ветвления. Обратите внимание, что формат поля подсказки зависит от реализации, и любая реализация ADAM должна работать правильно независимо от содержимого поля подсказки.

Команды перехода имеют поле назначения и 32-битную беззнаковую подсказку назначения перехода. В счетчик программ загружаются только нижние 32 бита значения в очереди, указанной полем назначения перехода. Поле подсказки назначения перехода предусмотрено для того, чтобы реализация могла запомнить последний адрес перехода. Обратите внимание, что формат поля подсказки зависит от реализации, и что любая реализация ADAM должна работать правильно независимо от поля подсказки содержание.

Инструкции подсказок — это пустые операции, которые предоставляют подсказки системе времени выполнения. Подсказка может зависеть от платформы или нет; эта информация кодируется в поле типа подсказки. Примерами подсказок являются директивы размещения данных, директивы предварительной выборки и директивы выхода потока. Подсказки, которые не распознаются средой выполнения, игнорируются.

Подробный список инструкций, поддерживаемых ADAM, и их описание см. в приложении D.

описания.

### Б.3 Формат возможностей

Формат возможностей, используемый ADAM [BGKH00], позволяет точно определять базу и границы из произвольной возможности с использованием front-padding для устранения небольшого количества накладных расходов на округление. Общий штраф за заполнение, вызванный форматом возможностей, ограничен менее чем 11,2% [BGKH00].

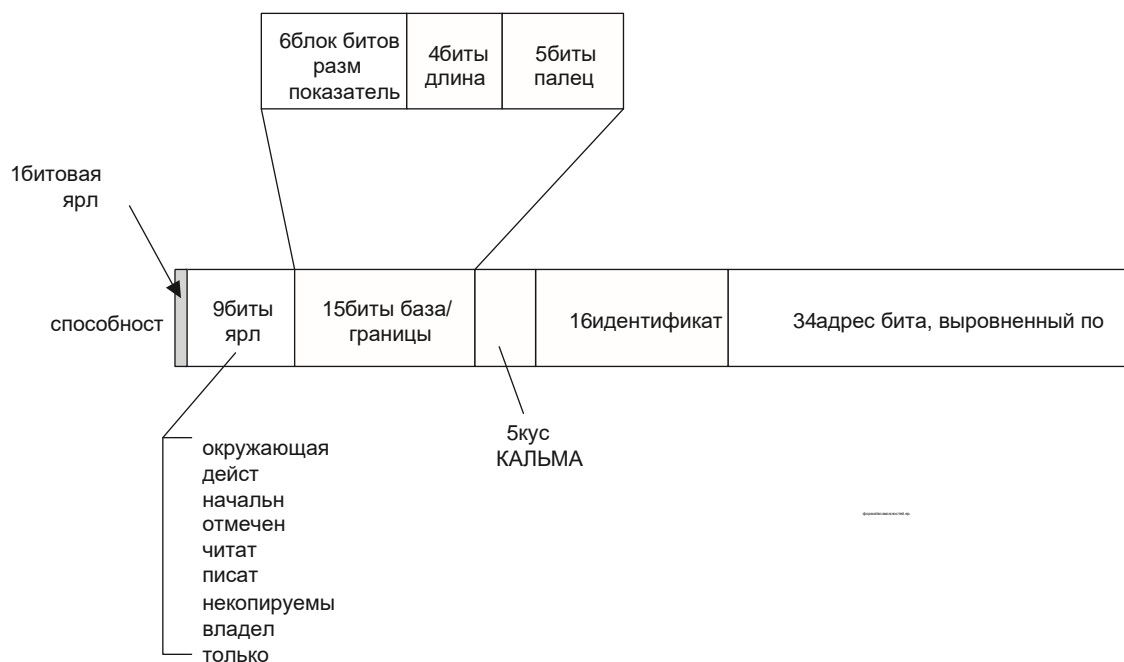


Рисунок В-4: Формат возможностей ADAM

Метод извлечения базы и границ из возможности с фронтальным заполнением довольно прост и может быть реализован непосредственно в оборудовании. Как показано на рисунке В-4, возможность включает в себя размер блока, длину, поле пальца и адрес. Сочетание размера блока и длины может использоваться для определения конца возможности; поле пальца используется для вывода местоположения начала возможности с учетом указателя на середину возможности. Размер блока из всех единиц (63 в данном случае, поскольку поле размера блока имеет длину 6 бит) является особым случаем, когда поле длины напрямую равно количеству слов в возможности. Эта уникальная структура была выбрана для упрощения аппаратной реализации, как описано в [BGKH00].

Метод извлечения базы и границ из возможности с фронтальным заполнением выглядит следующим образом и записан в виде псевдокода:

```

= значение поля размера блока
= длина значения поля
= значение поля пальца
= значение поля адреса

если(
    // , неизменяемы
    // и обновляются арифметическими операциями возможностей,
    // с проверкой, чтобы убедиться, что
    capabilities.beginning = ;
    возможность.длина = ;
    возможность.конец = возможность.начало + возможность.длина;
если(
    выдать исключение границ возможностей

    желаемые данные = * ;

еще
    // & – это побитовый оператор И
    возможность.начало = & ( ; возможность.длина = ;
    конечная.возможность = начальная.возможность +
    длина.возможности; желаемые данные = * ;

```

Единственными допустимыми операциями для возможности являются сложение и вычитание. Новый адрес, который получается в результате арифметической операции, легко вычислить:

```

= знаковое целое смещение, которое будет добавлено
= новый адрес
= старый адрес

= +

```

Метод пересчета поля `finger` возможности, над которой была выполнена арифметическая операция, выглядит следующим образом и записан на псевдокоде с синтаксисом битового поля Verilog:

```

= исходное поле пальца
= новое поле пальца

```

= знаковое целое смещение, которое будет добавлено

= значение поля длины блока if( )

= ;

еще

= & ( ;

Значение нового поля `finger` должно быть меньше значения поля `length`, но больше нуля; в противном случае должна быть отмечена ошибка. Эффективная аппаратная реализация приведенного выше расчета также приведена в [BGKH00]. Обратите внимание, что возможности не могут быть динамически изменены. Это означает, что поля `length` и `block size` никогда не должны изменяться после арифметической операции. Чтобы увеличить возможность, необходимо создать новую возможность и скопировать содержимое старой в новую.

Формат возможностей ADAM содержит явный идентификатор узла процессора, встроенный в поле адреса возможности. Размер поля идентификатора узла позволяет иметь до 65 536 процессоров в системе, но фактическое распределение возможностей на этих узлах остается за операционной системой. Все приложения ADAM могут работать на реализациях с любым количеством узлов от одного до 65 536, без требования распределения идентификаторов узлов, поскольку возможности непрозрачны для программиста, а процесс распределения зависит от реализации. Допустимые идентификаторы узлов могут даже изменяться динамически, при условии, что ОС внимательно следит за тем, чтобы узел был пустым перед деактивацией его идентификатора. Динамическое переназначение идентификатора может быть полезным в ситуациях, когда мониторы окружающей среды обнаруживают надвигающийся сбой или когда пользователи хотят выполнить горячую замену узлов для выполнения обновлений или обслуживания. Обратите внимание, что объем доступной памяти для запуска приложений зависит от количества узлов в системе, но адресное пространство довольно велико, поэтому пользователи редко сталкиваются с такой ситуацией.

Формат возможностей также включает ряд битов для управления памятью и безопасности. Эти биты:

**окружающая среда/данные:** указывает, относится ли возможность к пространству среды или к пространству данных. Обычно этот бит не следует изменять после создания возможности.



**только приращение:** указывает, что только положительные смещения от базы возможностей могут быть доступны **valid:** указывает, является ли возможность действительной. Попытка разыменовать недействительную возможность приводит к ошибке защиты.

**отмечено:** используется для сборки мусора. **read:**

указывает, что данные могут быть прочитаны из

возможности. **write:** указывает, что данные могут

быть записаны в возможность.

**некопируемый:** указывает, что для возможности разрешены только операции

извлечения из очереди; попытка скопировать возможность приведет к возникновению

исключения. **владелец:** если установлен бит владельца, биты чтения, записи и

некопируемости могут быть переопределены.

**начальный:** указывает, что эта возможность является основной рабочей копией. Для

возможностей в пространстве данных она отмечает конечную точку списка миграции.

Для возможностей в пространстве среды она также отмечает поток с этим установленным битом как единственную работоспособную копию.

**КАЛЬМАР:** Короткий квазиуникальный идентификатор. Короткое поле тега, которое содержит случайно сгенерированный номер идентификатора, назначенный во время распределения возможностей; при переносе возможностей это поле напрямую копируется. Использование этого поля снижает стоимость сравнений неравенства возможностей. [GBHK00]

## Б.4Uber-возможности и многозадачность

U<sup>ber</sup>-capability — это возможность, которая имеет доступ ко всему пространству памяти машины. Эта uber-capability используется потоками ядра для функций управления системой, поскольку ADAM не предоставляет режима супервизора или явных разрешений ядра в стиле Java. При включении питания каждый физический узел начинает выполнение кода в месте 0 в пространстве кода, а u<sup>ber</sup>-capability изначально помещается в q0. u<sup>ber</sup>-capability устанавливается равным размеру всей виртуальной памяти, доступной для этого узла, и

устанавливается бит владельца. Благодаря этому механизму код ядра, загруженный в место 0 в пространстве кода, может иметь доступ ко всему компьютеру. Этот код ядра также обычно является обработчиком исключений по умолчанию для узла.

Поскольку ADAM — это виртуальная машина, многозадачность на одной большой машине достигается путем разделения машины на меньшие группы физических узлов и запуска ADAM для каждой задачи, и каждый ADAM выполняет только одну задачу. Баланс нагрузки машины может быть установлен частично путем управления количеством узлов, к которым ADAM может получить доступ. Это ограничение доступа может быть достигнуто частично путем ограничения размера *uber-capability*.

## **Б.5Обработка исключений**

Исключения в ADAM по своей сути неточны. ADAM — это распределенная машина, которая запускает множество параллельных потоков; в этом сценарии нет четкого определения «одновременности». Подход ADAM к исключениям двоякий: во-первых, помечается результат каждого события, вызывающего исключение; во-вторых, сохраняется как можно больше локального состояния, относящегося к исключению, в момент обнаружения исключения.

Возможность исключения включена как часть состояния каждого потока. Эта возможность исключения инициализируется при создании потока, чтобы указать на объект обработки исключений по умолчанию (обычно объект, определенный ОС), и может быть переопределена пользователем в любое время. Обработчик по умолчанию вызывается в случае, если определенный пользователем обработчик исключений недействителен. Пользователи могут использовать этот механизм для построения цепочек обработчиков исключений, как показано на рисунке В-5.

Исключения обрабатываются на основе узла процессора. Когда поток сталкивается с  
исключением,  
обработчик исключений немедленно планируется для запуска на этом узле и блокируется как единственный работающий поток до тех пор, пока исключение не будет разрешено. Обработчик исключений проверяет регистр состояния процессора и регистр идентификатора исключенного контекста, чтобы определить источник исключения. Затем,

при необходимости, для связи с исключенным потоком используется протокол, определенный ОС. Обычно этот протокол включает в себя принудительное удаление исключенного контекста из файла очереди обработчиком исключений и его изменение в пространстве окружения исключенного контекста. Этот процесс может занять тысячи циклов. Исключения должны быть редкими событиями, и пользователи должны избегать использования механизма исключений для чего-либо, кроме обработки исключений. Другими словами, их следует избегать в целом как механизма для реализации API или аппаратных интерфейсов.

Вместо этого пользователям рекомендуется использовать сопоставления очередей и расширения опкодов в форме, похожей на инструкции ALLOCATE или SPAWN. Расширения опкодов могут быть реализованы с использованием механизма обработчика недопустимых опкодов, описанного ниже.

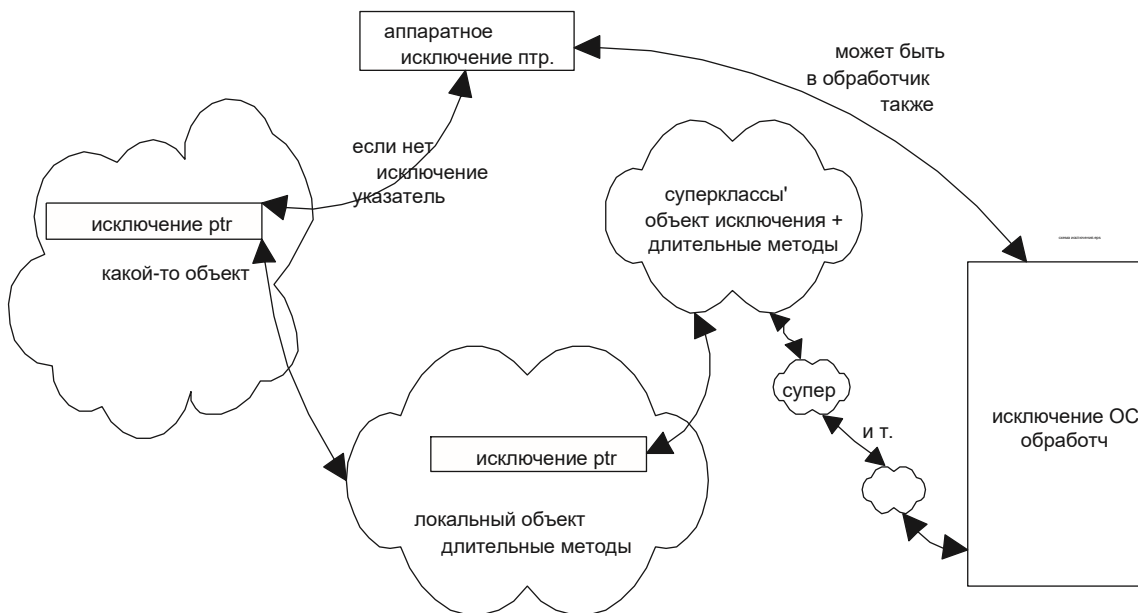


Рисунок В-5: Обзор обработки исключений

Исключения недопустимых кодов операций обрабатываются особым образом, аналогично PALcode архитектуры Alpha. Недопустимый код операции отправляется в таблицу поиска в памяти, которая имеет жестко зашитый адрес, и поток управления передается процессору микрокода конкретной реализации, который имеет доступ ко всем

локальным состояниям. Процессор микрокода может быть таким же простым, как выделенный идентификатор контекста на ADAM плюс расширения набора инструкций. Код, который выполняет процессор микрокода, хранится в зарезервированном месте в памяти ядра; это позволяет эмулировать инструкции, реализованные в будущих версиях архитектуры, с помощью программных исправлений, установленных ОС. В режиме эмуляции процессор ведет себя так, как будто он остановился, и ошибки в режиме эмуляции приводят к неопределенному поведению. Я рекомендовал, чтобы поведением по умолчанию для недопустимого кода операции был эмулируемый THROW инструкция.

# Приложение С

## Подробности Q-машины

Новичок пытался починить сломанную машину Lisp, выключая и включая питание.

Найт, увидев, что делает студент, строго сказал: «Вы не можете починить машину, просто включив ее снова и снова, не понимая, что именно не так». Найт выключил и снова включил машину.

Машина заработала.

*—Традиционный III Коан*

В этом приложении приводятся многие детали, опущенные в главе 5 в целях ограничения основного текста диссертации только функциями и вопросами, относящимися к миграции. В частности, в этом разделе обсуждаются детали реализации PQF, реализации сетевого интерфейса и транспортного протокола, а также топология сети, используемая в реализации Q-Machine и системном симуляторе ADAM.

### С.1 Подробности реализации файла очереди

В этом разделе обсуждаются некоторые важные детали реализации PQF, используемые системным симулятором ADAM. Эта часть оборудования, возможно, является наиболее сложным компонентом для реализации в реализации Q-Machine, поэтому она заслуживает некоторого исследования в контексте этого диссертация.

### С.1.1 Физический дизайн

В основе VQF лежит физический файл очереди (PQF), который напрямую реализует архитектурно неопределенное количество очередей. Высокоуровневый эскиз PQF можно увидеть на рисунке С-1. PQF напрямую присоединен к вычислительным блокам. Размер PQF должен быть установлен деталями целевого процесса реализации; однако для хорошей однопоточной производительности PQF должен воплощать по крайней мере 128 очередей, доступных для одного контекста. PQF имеет структуру, похожую на многопортовый регистровый файл, и он способен заменять целую очередь в Queue-Cache (QC) и из него за один цикл. Пустые очереди не заменяются в QC; вместо этого они просто помечаются как пустые и не потребляют дополнительную полосу пропускания или пространство. Подсистема памяти содержит специальное оборудование для ускорения маркировки и замены пустых очередей. Хороший компилятор сделает так, чтобы все очереди потоков были пустыми при остановке выполнения, так что мертвые потоки потребляют минимальный объем памяти, пока они не будут удалены сборщиком мусора.

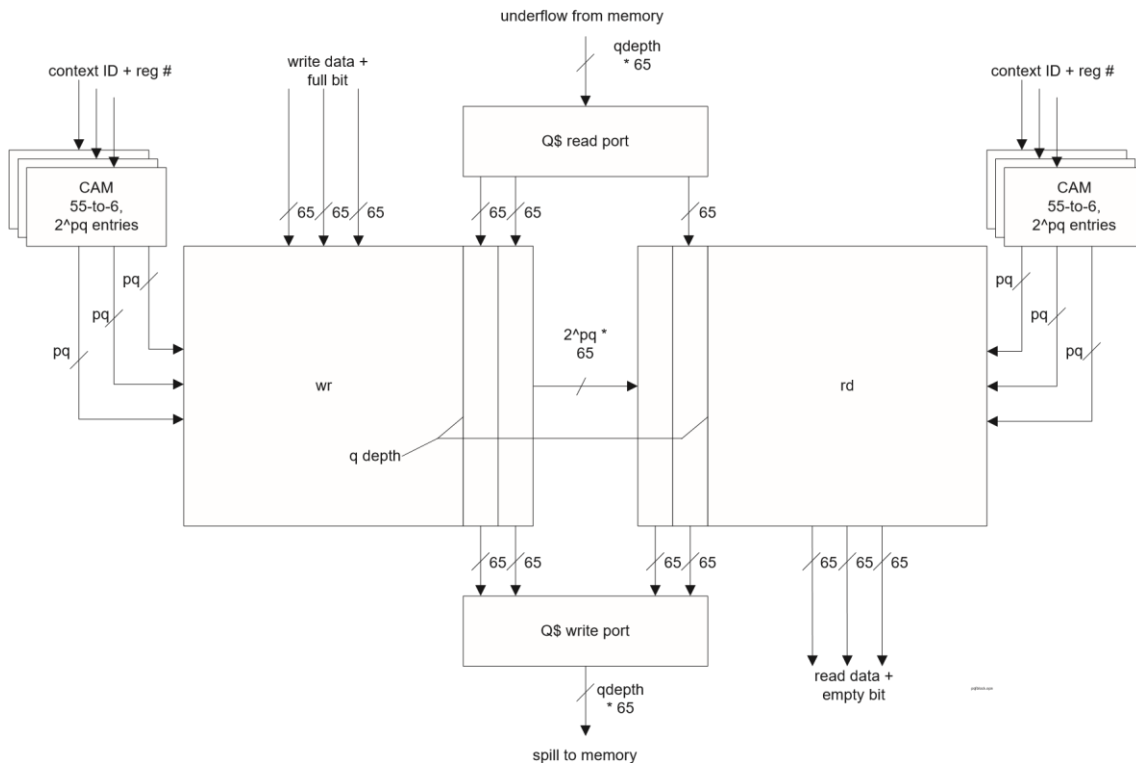


Рисунок С-1: Реализация VQF с 3 портами записи и 3 портами чтения.  $pq = \#$  физических регистров. Подробности Q-кэша опущены для ясности.

QC имеет структуру, похожую на кэш памяти; когда он переполняется, строки кэша стратегически записываются в основную память. Тот факт, что каждая очередь в системе имеет некоторое место в памяти, зарезервированное для ее хранения, является функцией, которая используется механизмом GC для очистки после мертвых потоков или для миграции объектов.

Физический файл очереди на самом деле не занимает значительно больше места, чем обычный многопортовый регистровый файл. Причина этого в том, что регистровый файл доминирует на проводах; активная область транзистора под ячейкой регистрового файла составляет малую часть области, выделенной для проводов.

На рисунке С-2 показана элементарная ячейка PQF с 3 портами чтения и 3 портами записи с достаточным количеством проводов Q-кэша для управления очередью глубиной 4.

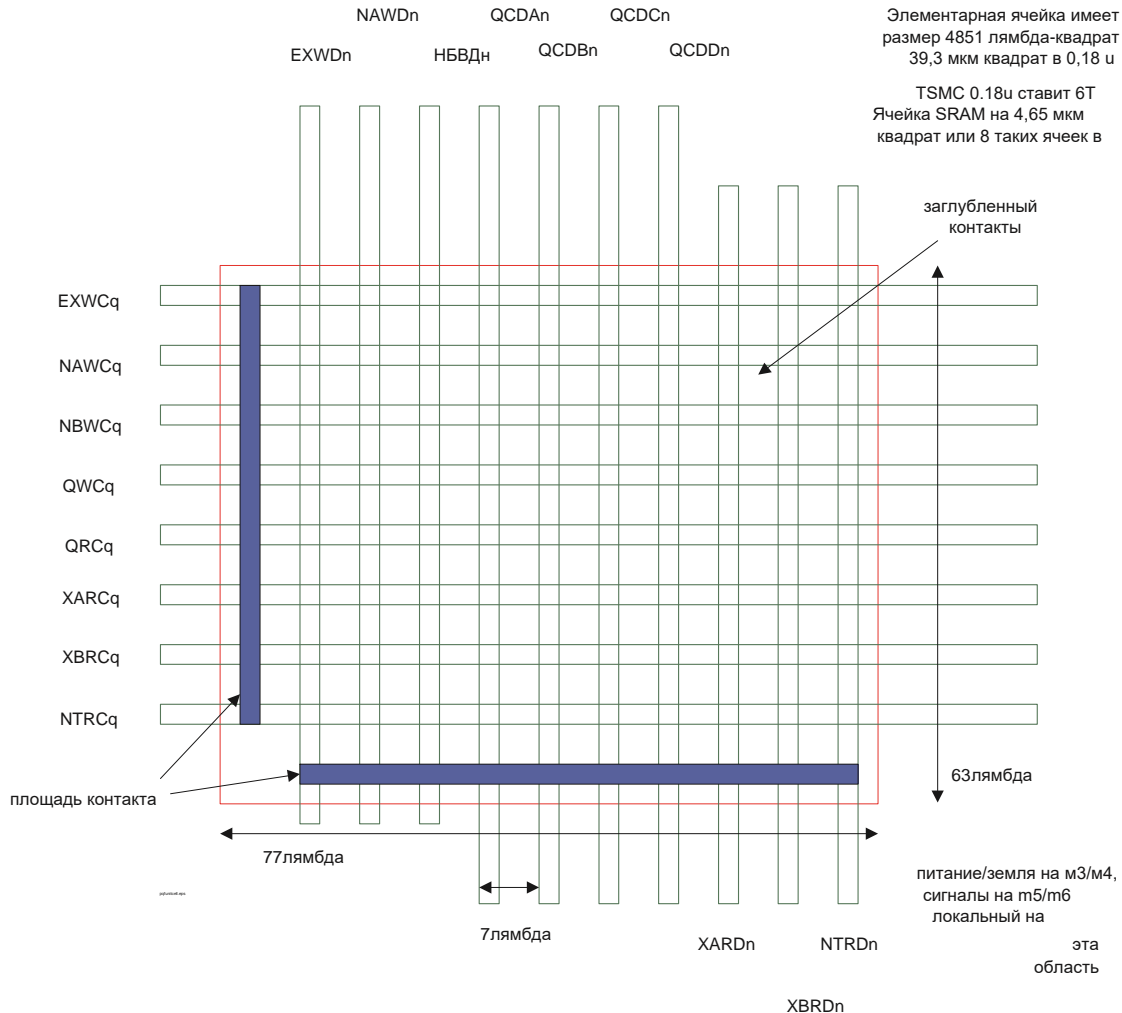


Рисунок С-2: Элементарная ячейка PQF.

Шаг проводки основан на числах, взятых из руководства по процессу TSMC 0,18 м [Corb]. Требования к проводке для ячейки PQF потребляют только 4851, используя провода с минимальным шагом M5/M6. Для сравнения, площадь ячейки 6-T SRAM в процессе TSMC 0,18 м составляет

574 , что позволяет разместить восемь таких ячеек под ячейкой PQF. Для лучшей производительности можно использовать более толстые провода с большим расстоянием, тем самым увеличивая площадь под ячейкой для реализации фактической структуры Qлогика хранения и управления.

Следовательно, реализация PQF, которая имеет относительно мелкие очереди (от 4 до 8 глубиной), может быть реализована в объеме, в два раза превышающем объем обычного файла регистров с аналогичным количеством портов. По мере развития технологии процесса будут включены даже более глубокие очереди за счет большего количества или более быстрых проводов, необходимых для обмена с Q-кэшем. Подходящая, высокопроизводительная асинхронная конструкция FIFO описана в [MJC 99] и [MJCL97]. Эти FIFO глубиной 17 надежно работали с пропускной способностью 1,7 гигабайта элементов данных в секунду в 0,6-метровом процессе CMOS. Варианты этой конструкции были исследованы автором, но не представлены здесь в интерес краткости.

Аналогичная идея реализации VQF, описанная здесь, — это файл регистра именованного состояния (NSRF). [ND91] [ND95] NSRF — это файл регистра с автоматизированным механизмом для сброса и заполнения контекстов потоков. Он использует номера идентификаторов контекста для уникальной идентификации потоков и память CAM для сопоставления отдельных записей файла регистра с их надлежащими контекстами. В отличие от VQF, NSRF сбрасывает свое состояние непосредственно в кэш данных процессора. Q-Machine этого не делает, поскольку на Q-Machine нет кэша данных, и даже если бы он был, сочетание необходимости добавления дополнительного порта чтения/записи в D-Cache и проблем с загрязнением кэша представило бы веские аргументы в пользу наличия отдельного Q-cache. Хотя VQF вводится в первую очередь для поддержки разьединенного физического-логического отображения процессоров на потоки, интересно отметить, что NSRF действительно обеспечил небольшое (от 9% до 17%) ускорение параллельного и последовательного выполнения программ. Также следует отметить, что



файлы регистров в стиле кэша, такие как NSRF и VQF, обеспечивают более высокую общую загрузку файла регистра: было продемонстрировано, что NSRF имеет на 30–200 % лучшую загрузку, чем обычный файл регистра. [ND95]

Блок MAP отвечает за определение того, сопоставлена ли очередь с другим контекстом. Блоку MAP выдается запрос на обнаружение сопоставления очереди в момент выдачи инструкции, что дает ему всю задержку конвейера машины для выполнения этой работы. Операция MAP потенциально сложна и может стать причиной множества остановок, если машина спроектирована неправильно.

Причина, по которой блок MAP нужно декодировать только для целей записи, заключается в том, что единственными допустимыми отображениями очереди, разрешенными на Q-Machine, являются прямые отображения. Другими словами, невозможно создать отображение, которое «вытягивает» данные из другого контекста; вместо этого можно только вводить данные в целевой контекст. Как видно из диаграммы, функция MAP, таким образом, вызывается как для входящих записей из NI, так и для локальных результатов из блока ALU/MEM. Это сохраняет низкую задержку чтения из VQF, в то же время давая функции MAP время для выполнения трансляции для записей.

Напомним, что идентификатор контекста для потока на самом деле является возможностью, которая указывает на область хранения для резервного хранилища потока и локального хранилища данных. Эта возможность имеет разрешения, установленные таким образом, что пользовательский процесс не может разыменовывать эту возможность и использовать ее в качестве указателя памяти, но ОС и функция MAP имеют доступ к этой информации в любое время. Обратитесь к 3.2.3 для обзора формата адреса возможности Q-Machine. Учитывая это, базовый алгоритм для блока MAP выглядит следующим образом:

Если поле Proc ID контекстного ID не равно Proc ID локального процессора, отправьте запись в NI

В противном случае обратитесь к внутреннему кэшу, который регистрирует наличие сопоставления в указанной очереди для указанного контекста. Если сопоставления нет, передайте запись в VQF. Если сопоставление есть, обратитесь к таблице сопоставлений, чтобы обнаружить правильное сопоставление, и отправьте данные в NI для маршрутизации (даже если это сопоставление-к-себе). Пометьте очередь как заполненную и заблокируйте поток, пока NI не сообщит об успешной доставке данных

Кэш присутствия карты используется для ускорения типичного случая, когда сопоставление отсутствует. В памяти может храниться больший кэш присутствия карты, чем кэш с битами присутствия и фактическими сопоставлениями. В случае переполнения таблицы сопоставления должен произойти поиск в резервной таблице, и машина зависает. Кроме того, в случае, если сопоставление существует, можно потратить несколько дополнительных циклов, чтобы извлечь сопоставление из памяти. Возможно, также будет поддерживаться небольшой кэш сопоставлений, если будет установлено, что сопоставление является серьезным узким местом.

### **С.1.2 Государственная машина**

Запросы на заполнение из PQF генерируются в ответ на пропущенные запросы чтения и записи выданными инструкциями. Для запросов на чтение строка-заполнитель помечается в PQF, и запрос на заполнение выдается в память среды. Для запросов на запись все сложнее. Если очередь никогда не создавалась ранее в контексте, пустая строка в PQF просто преобразуется в полноценную строку чтения/записи с помеченным грязным битом. Если есть существующий заполнитель чтения, эта строка преобразуется в строку только для записи с данными записи, ожидая заполнения слиянием с уже выданным запросом на заполнение. В противном случае, если в PQF есть место, создается строка только для записи и выдается запрос на заполнение слиянием чтения. Если в PQF нет места для запроса записи, делается запрос на пустую строку, и как только он удовлетворен, создается строка только для записи с запросом на заполнение слиянием.

Запросы на сброс из PQF генерируются в ответ на следующие события: переполнение PQF,

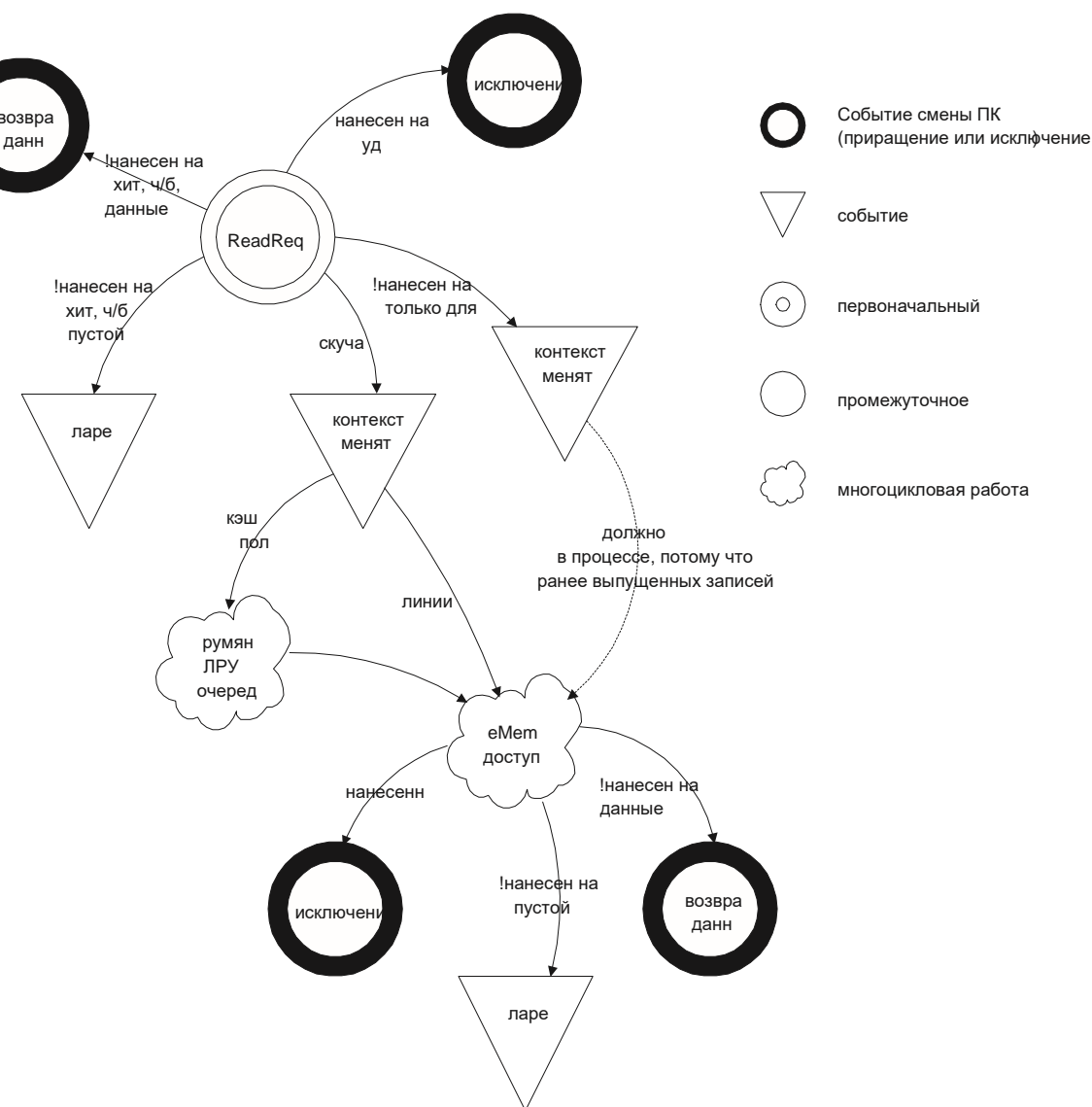


Рисунок С-3: Блок-схема ответа на запрос чтения PQF, переполнение планировщика и миграция. В случае переполнения PQF выбирается строка LRU и загружается из файла очереди. Следующие два случая не документированы на рисунках С-3 и С-4. В случае переполнения планировщика планировщик больше не может отслеживать все вспомогательные данные, связанные с контекстом, и он хочет удалить все это в память среды. В случае запроса миграции все состояние потока должно быть снова удалено в память, но, кроме того, некоторые данные могут быть перенаправлены через сетевой интерфейс до завершения удаления. Чтобы строки только для записи с merge-fills и грязные строки обрабатывались правильно в месте назначения миграции, перенесенная строка должна переместиться со своими тегами и вставиться в очередь назначения вместе с любыми

ожидающими запросами, связанными с типом тега. В конечном итоге механизм заполнения будет работать, чтобы получить данные из исходного контекста с помощью указателей пересылки, но в то же время вычисления могут возобновиться. Любые выполняемые заполнения для исправления плейсхолдеров или строк только для записи локально разрешается завершить, но возвращаемые данные отбрасываются. Это приемлемо, поскольку строки только для записи удаляются в память среды с запросом на слияние (и, как отмечалось ранее, строки только для записи, напрямую отправленные в пункт назначения, вставляются с запросами на слияние). Конечно, плейсхолдеры можно просто отбросить.

Следующее состояние также сохраняется в строке PQF или должно быть синхронизировано с резервным хранилищем памяти среды при выводе из эксплуатации в дополнение к необработанным данным очереди:

Созданные биты

Резидентские биты

Отображенные биты

Контекст назначения карты

Карта назначения VQN

Флаг очереди источника карты (также влияет на сетевой интерфейс)

Сопоставить исходную очередь с сестрой (также влияет на сетевой интерфейс)

Флаг очереди источника карты и значения сестры очереди источника карты должны сообщаться сетевому интерфейсу всякий раз, когда очередь заменяется в PQF или из него. Это происходит потому, что исходные данные потока удаляются из входящего сетевого пакета реализацией транспортного уровня. Таким образом, когда установлен бит флага очереди источника карты, транспортный уровень должен сохранить исходный контекст потока из сетевого пакета и сгенерировать запрос на запись в PQF как для поступающих данных, так и для идентификатора контекста отправителя этих данных.

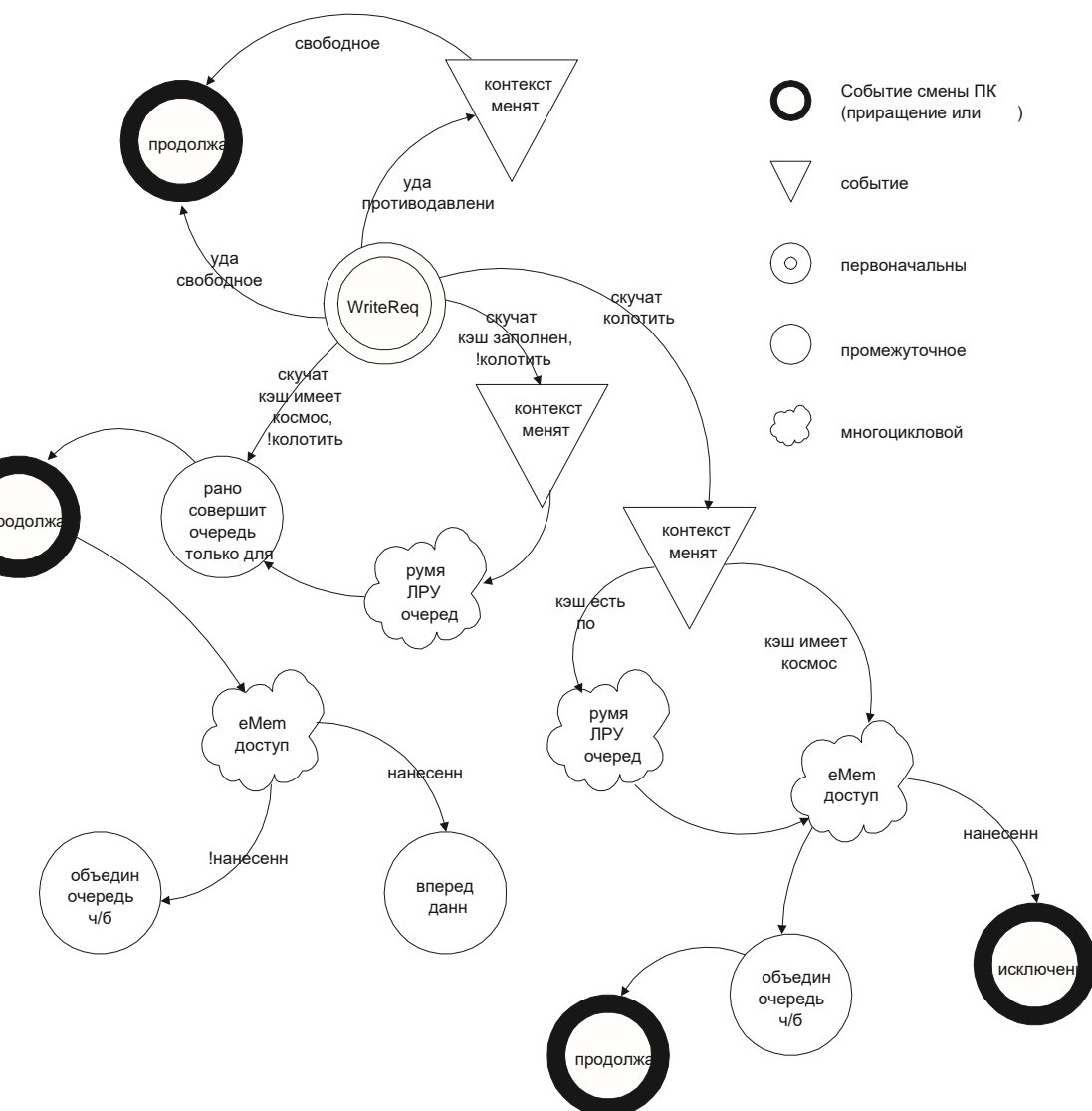


Рисунок С-4: Блок-схема ответа на запрос записи POF

## С.2 Сетевой интерфейс

Сетевой интерфейс реализует на аппаратном уровне функции, аналогичные физическому, каналному, сетевому и транспортному уровням из сетевого стека OSI 7-level. В этой реализации физический и каналный уровни объединены и называются PNY, а сетевой и транспортный уровни объединены и называются XPORT. Такое снижение абстракции было выбрано, во-первых, потому, что сетевой протокол для реализации ADAM очень прост, а во-вторых, потому, что есть сильная мотивация уменьшить задержку путем исключения ненужной буферизации и инкапсуляции пакетов. При этом сетевой интерфейс предоставляет следующие услуги:

абстрактный и модульный интерфейс от ядра процессора до физического уровня сети надежная доставка данных генерация остановок ядра процессора при перегрузке сети идемпотентная доставка данных сквозной путь к узлам памяти, который обходит задержки XPORT- и RHY-уровней поточная доставка данных кольцевой путь для межпоточных данных на одном и том же узле процессора

Сетевой интерфейс предполагает, что RHY не несет ответственности за доставку пакетов и буферизацию пакетов. Следовательно, уровень XPORT должен реализовать надежную доставку и попытаться всегда гарантировать пространство для прибывающих пакетов. Обсуждение того, как реализованы топология и маршрутизация на уровне RHY, можно найти в разделе C.3. Читателям рекомендуется обратиться к этому разделу, если они не знакомы с сетями с коммутацией каналов и маршрутизацией через червоточины и ненадежной (но быстрой) маршрутизацией.

Обзор реализации сетевого интерфейса можно найти на рисунке C-5. Данные, поступающие из ядра процессора, сначала имеют свои заголовки источника и назначения, а затем маршрутизируются маршрутизаторами cut-through и loopback. Путь данных в этой точке маршрутизирует информацию источника, назначения и данных параллельно, чтобы сократить задержку.

Маршрутизатор cut-through и loopback просто распознает адреса, предназначенные для локального процессорного узла или памяти, и передает эти данные напрямую к месту назначения, минуя уровни XPORT или RHY. Данные, проходящие через тракты cut-through и loopback, всегда гарантированно имеют длину в одно слово. Если пункт назначения cut-through или loopback не может принять данные, он должен отправить отправителю сигнал о задержке, и в маршрутизаторе cut-through и loopback должна быть достаточная буферизация для компенсации времени прохождения сигнала о задержке. Все данные, не предназначенные для

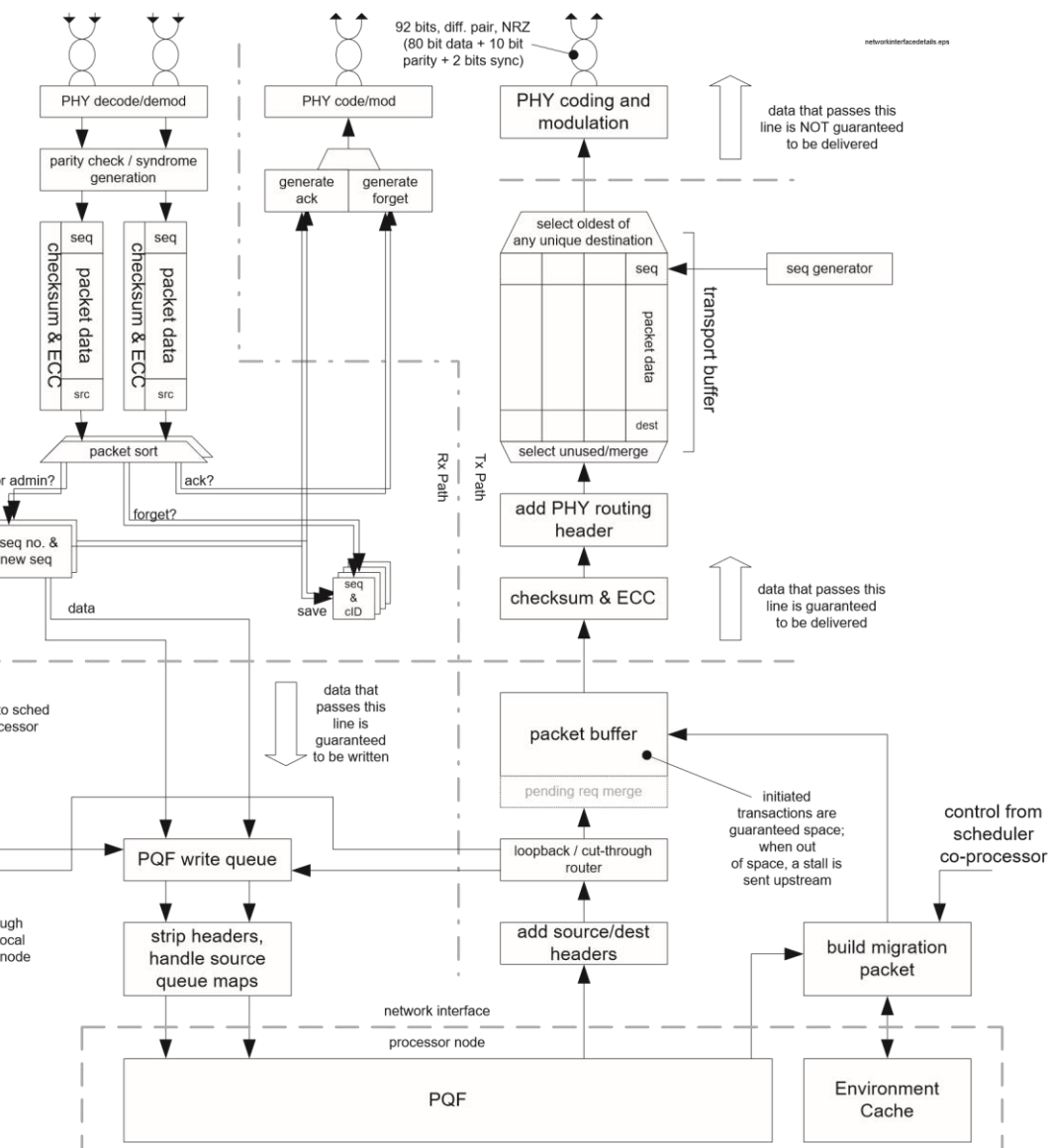


Рисунок С-5: Подробная информация о сетевом интерфейсе. Сквозные или петлевые пути отправляются в исходящий буфер пакетов.

Буфер исходящих пакетов отвечает за регулирование потока данных от процессорного узла и менеджера миграции, поступающих на уровень XPORT. Он должен быть достаточно большим: достаточно большим, чтобы вместить несколько пакетов максимальной длины. Он также должен быть достаточно гибким, поскольку большинство пакетов будут иметь размер либо в пару слов, либо в пару сотен слов. Наконец, буфер пакетов должен реализовать следующий контракт с менеджером миграции и узлом процессора: как только данные заданной длины были приняты для передачи в буфер, он должен иметь возможность принять все эти данные. Если он не может, он должен отклонить любые данные от

отправителя, и отправитель должен повторить отправку данных. В случае процессорного узла задержка приведет к тому, что отправляющий поток будет удален в список планировщика, и повторная попытка произойдет, когда планировщик снова попытается запустить застрявший поток. В случае менеджера миграции сопроцессор планировщика отвечает за реализацию некоторой схемы отсрочки и повторной попытки в программном обеспечении. Буфер пакетов также может опционально реализовать слияние пакетов для данных, отправляемых в один и тот же пункт назначения. Он всегда должен сохранять временной порядок данных после слияния и не может объединять атомарные операции EXCH.

Данные, принятые в буфер пакетов, затем пересылаются на уровень XPORT, но только при наличии свободного места в буфере XPORT. Уровень XPORT добавляет требуемую контрольную сумму, ECC и заголовки маршрутизации RNY-уровня перед сохранением данных в транспортном буфере. Уникальный порядковый номер процессорного узла также добавляется к сообщению при сохранении в транспортном буфере для реализации надежной, идемпотентной и упорядоченной доставки.

Протокол, используемый для надежной идемпотентной доставки, опирается на уникальные порядковые номера и маркеры подтверждения/забывания. Это протокол, разработанный Джереми Брауном и Дж. П. Гроссманом из лаборатории искусственного интеллекта Массачусетского технологического института с дополнительным уровнем нумерации, гарантирующим доставку сообщений в порядке их следования. [GB02] На рисунке С-6 показан упрощенный протокол без доставки в порядке следования. Каждому сообщению в отправителе назначается уникальный порядковый номер. Этот номер может быть гарантированно уникальным, если просто использовать очень большой (64-битный) счетчик и увеличивать его один раз для каждого пакета данных. Отправитель запоминает пакет данных и порядковый номер даже после отправки пакета. Как только получатель получает пакет, он запоминает порядковый номер и идентификатор контекста отправителя и передает часть данных по назначению в узле. Затем получатель как можно скорее возвращает отправителю пакет ACK с порядковым номером в качестве полезной нагрузки; когда отправитель видит пакет ACK, он знает, что может спокойно забыть о буферизованном пакете; надежная доставка произошла. Однако, если по истечении некоторого периода ожидания ACK не получен, отправитель должен повторно отправить свой пакет данных.



Если ACK не был получен из-за того, что путь ACK был заблокирован или поврежден, то возникает риск двойной записи данных. Однако это предотвращается, поскольку получатель отслеживает порядковый номер отправителя; любой входящий пакет с неуникальным порядковым номером от конкретного узла процессора отбрасывается, и генерируется другой ACK. Пакет FORGET требуется получателю, чтобы он знал, когда он может удалить порядковые номера и прекратить отправку пакетов ACK. Таким образом, всякий раз, когда отправитель получает пакет ACK, он немедленно возвращает пакет FORGET источнику пакета ACK с порядковым номером пакета ACK в качестве

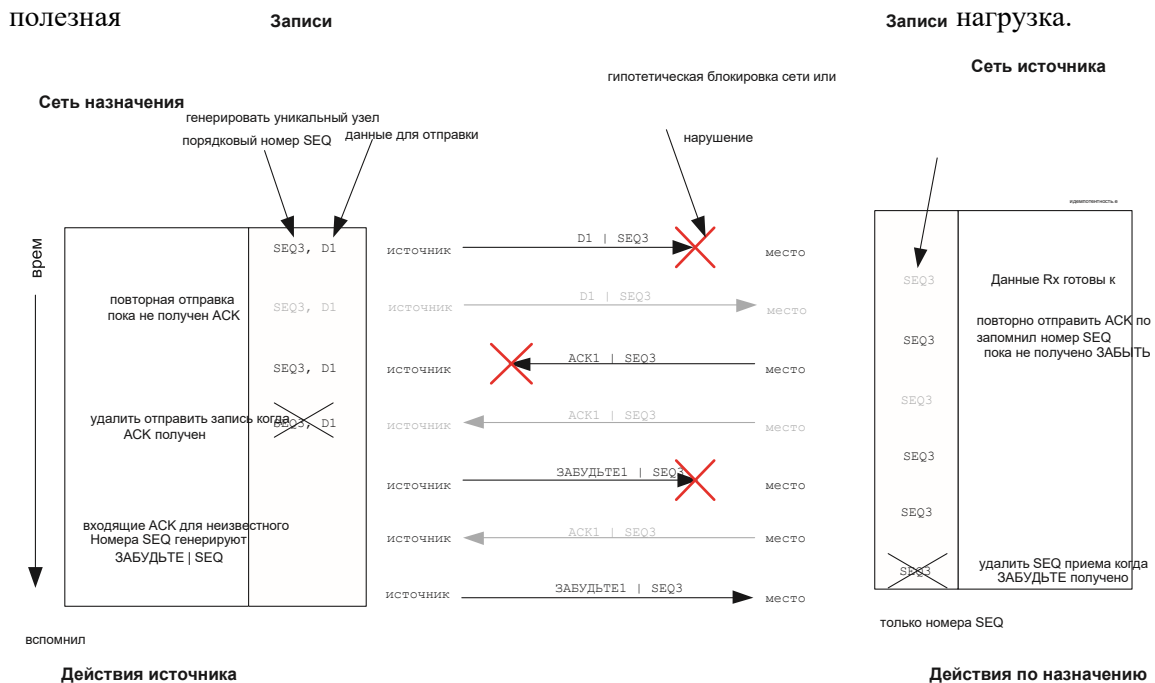


Рисунок С-6: Идемпотентность и надежный протокол доставки данных в деталях для одной транзакции. Линии серого цвета — это линии «повторов», которые не произошли бы в идеальных условиях.

Базовый протокол, описанный выше, не гарантирует доставку пакетов в нужном порядке к месту назначения. Пакеты могут быть переупорядочены из-за того, что любой пакет в последовательности пакетов может не быть доставлен с первой попытки. Моя настройка протокола заключается в том, чтобы включить дополнительный номер очереди в каждый пакет. Номер очереди начинается с нуля и увеличивается каждый раз, когда пакет отправляется для данного порядкового номера. Задача получателя — воссоздать исходный порядок пакетов с использованием номеров очереди. Дополнительное сообщение FORGET

CONNECTION требуется для сигнализации о том, когда номер последовательности может быть забыт. Номера последовательности могут быть забыты только по истечении достаточного времени, чтобы гарантировать, что последний пакет, отправленный в протоколе, был либо успешным, либо неудачным. Это легко определить, поскольку сеть является коммутируемой, а время доставки изначально ограничено. Доставка пакета считается неудачной, если подтверждение не получено по истечении периода времени, равного времени на передачу и прием плюс время накладных расходов на подтверждение.

Сетевой интерфейс структурирован так, чтобы иметь один выделенный порт передачи пакетов данных, один выделенный порт передачи пакетов ACK и FORGET и два порта приема. Эта структура помогает регулировать поток данных в сети и гарантировать, что пакеты ACK и FORGET (которые меньше пакетов данных и, следовательно, могут быть отправлены с большей скоростью) имеют больше шансов попасть в сеть. Эта структура также помогает смягчить конфликт портов на приемниках, ограничивая пиковую скорость ввода сообщений строго ниже пиковой скорости приема сообщений. Более подробную информацию о том, как структурированы заголовки маршрутизации, а также о количестве и типах проводов, используемых для интерфейса с сетью, см. в разделе С.3.

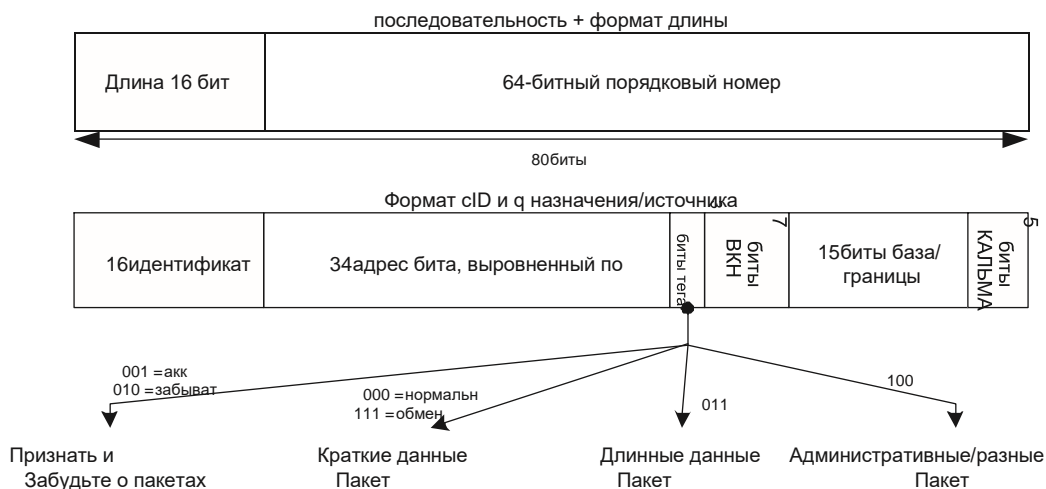




Рисунок С-7: Подробности форматов пакетов. Обратите внимание, что в заголовках cID назначения/источника и очереди очень важно, чтобы идентификатор процессора находился в MSB и был совмещен с полем адреса, поскольку реализации могут помещать биты между полями адреса и PID для увеличения количества маршрутизируемых узлов процессора или для увеличения объема памяти на узел.

Формат сетевых пакетов задокументирован на рисунке С-7. Самая важная информация, которую можно почерпнуть из этой диаграммы, заключается в том, что пакеты ACK и FORGET являются самыми короткими пакетами, и что сокращенный «короткий пакет данных» доступен для очень распространенного особого случая, когда пакет содержит полезную нагрузку из одного слова. Короткий пакет данных на 20% короче длинного пакета данных длиной один, поскольку он объединяет поле ECC и порядковый номер в одно и то же слово и полностью удаляет поле длины. Поле ECC/контрольной суммы может быть короче для этих пакетов, поскольку для контрольной суммы и исправления требуется меньше данных.

### С.3 Топология сети и реализация

Топология сети и ее конкретные параметры реализации могут свободно меняться в зависимости от конечных требований пользователя. Протокол XPORT, обсуждаемый в разделе С.2, предполагает, что сеть является коммутируемой, т. е. в ней нет буферизации или переупорядочивания, и что она довольно ненадежна; с точки зрения отправителя конкуренция за порты и ресурсы маршрутизации неотличима от аппаратных сбоев. Эта ненадежность не так плоха, как кажется; в условиях небольшой нагрузки соединения устанавливаются быстро и надежно, а производительность соединения постепенно ухудшается по мере увеличения перегрузки. Однако при правильно спроектированном транспортном протоколе это явление можно использовать в качестве обратной связи для

регулирования скорости вставки сообщений. Сеть также довольно устойчива к аппаратным сбоям, поскольку она изначально спроектирована для того, чтобы справляться с такими сценариями. Симулятор системы ADAM реализует сетевую топологию, основанную на сети METRO, подробно описанной [DeH93] и [WC01].

Главным преимуществом сетей с коммутацией каналов по сравнению с сетями с маршрутизацией пакетов является производительность задержки и простота реализации. На маршрутизаторах не требуются буферы для обработки конфликта портов: сообщение просто отбрасывается, и отправитель несет ответственность за повторную отправку сообщения. Это связано с тем, что сеть не должна гарантировать доставку сообщения. Кроме того, при правильном выборе топологии сети маршрутизация может происходить на скоростях распространения волн, так что задержка в проводах, даже на коротких участках, является доминирующим компонентом задержки сети.

Одним из недостатков сетей с коммутацией каналов является то, что сеть очень неэффективна с точки зрения пропускной способности, если минимальное время, необходимое для установления соединения, велико по сравнению со временем, необходимым для доставки данных сообщения. Такой сценарий может возникнуть в компьютере размером с комнату, где скорость электромагнитных волн в медном волноводе заставляет минимальное время для установления соединения составлять несколько десятков или сотен периодов тактовой частоты процессора. Это особенно болезненно в случае, если соединение блокируется конфликтом маршрутизаторов вблизи пункта назначения, поскольку ресурсы маршрутизации были израсходованы и заблокированы по всему телу сети на протяжении всего соединения. Наихудший сценарий возникает, когда несколько отправителей пытаются связаться с одной удаленной точкой доступа в глубокой сети с коммутацией каналов, в результате чего несколько сообщений легко маршрутизируются через тело сети, потребляя ресурсы, только чтобы быть потерянными вблизи пункта назначения. Андре ДеХон из Калифорнийского технологического института предположил, что гибридная буферизованная пакетная и коммутируемая сеть может быть хорошим решением в этих условиях: можно вставить станции буферизации пакетов, которые хранят ограниченное количество пакетов (и могут свободно отбрасывать лишние пакеты) в промежуточных «контрольных точках» вдоль сети. В сценарии с точкой доступа сообщения легко маршрутизируются через большую часть сети и хранятся в контрольной точке,

ближайшей к точке доступа, и только сегмент сети между точкой доступа и ближайшей контрольной точкой страдает от дегенеративной перегрузки. Расстояние между контрольными точками и глубина буферов контрольных точек являются параметрами, которые сильно зависят от технологии реализации и, в частности, от соотношения времени распространения информации к временной длине среднего сообщения.

Другой метод снижения стоимости коммутации каналов, предложенный Дж. П. Гроссманом из MIT AI Lab, заключается в использовании маршрутизации через червоточину. Можно построить сеть с маршрутизацией через червоточину, используя те же быстрые структуры маршрутизаторов и протоколы, что и сеть с коммутацией каналов, но вместо того, чтобы удерживать канал, пока получатель не разорвет его с подтверждением, соединение разрывается по мере маршрутизации хвоста сообщения. Этот метод маршрутизации требует повторного установления отдельных маршрутов ACK и FORGET, но эта цена относительно невелика. Надежный протокол доставки и идемпотентности, описанный в разделе С.2, может обрабатывать заблокированные пакеты ACK и FORGET. Кроме того, полезная нагрузка данных доставляется в первом пакете, прибывающем в пункт назначения, поэтому даже если пакетам ACK и FORGET требуется некоторое время, чтобы пройти через систему, эффективная задержка доставки все равно является ценой одностороннего путешествия.

Рекомендуемая топология сети для реализаций ADAM — это гибридная топология, аналогичная предложенной в [DeH90]. Топология встроенной или локальной сети должна быть сетью radix-4 dilation-2 bidelta, как описано в [DeH93]. Для сетей вне чипа или сетей с большим расстоянием в более крупных системах требуется некоторое уточнение полосы пропускания для масштабируемости. Базовые компоненты маршрутизатора, разработанные для сетей внутри чипа, могут быть повторно использованы для реализации более масштабируемой топологии fat-tree, как описано в [DeH90] и в [WC01]. Параметры, принятые в реализации 2010 года, описанной в разделе С.2, подразумевают, что вне чипа сеть может работать на скоростях 2-4 ГГц с двойной тактовой частотой. Предполагая, что для представления одного flit требуется 92 бита, 46 дифференциальных пар могут передавать полное 80-битное слово плюс ECC, синхронизацию и тактовую частоту в каждом цикле процессора. Вероятно, в 2010 году на корпусе будет достаточно контактов для реализации десятков таких высокоскоростных соединений на чип.

Наконец, рекомендуется, чтобы реализация использовала единую частотную ссылку и мезохронную (нечувствительную к фазе) схему синхронизации для всей машины. Эта единая ссылка может иметь встроенную избыточность или вспомогательные резонаторы, распределенные по всей машине, чтобы предотвратить расплавление или сбой электросети из-за индуктивной отдачи в результате сбоя часов. Реализация потребует начальной фазы самокалибровки, когда приемники определяют оптимальную фазу выборки, и, возможно, даже потребует периодических (порядка минут или часов) повторных калибровок, когда машина останавливается на микросекунду или две, чтобы компенсировать изменения свойств материала с течением времени и температуры. Главное преимущество использования мезохронной схемы источника одной частоты заключается в том, что можно удалить время разрешения метастабильности из бюджета синхронизации сети, а второстепенное преимущество заключается в том, что это упрощает реализацию физического уровня, поскольку плезиохронные реализации требуют некоторой сложности для обработки случая, когда интегрированная ошибка частоты приводит к потере цикла между узлами. Влияние разрешения метастабильности и синхронизации на задержку узла маршрутизатора не следует недооценивать, особенно если маршрутизатор работает на скоростях, близких к скорости распространения волн. В чипе маршрутизатора SGI SPIDER, используемом суперкомпьютерами Origin 2000 [Gal96], 17,5 нс из общей задержки от вывода к выводу в 40 нс сжигается в синхронизаторе. Проблема с метастабильностью заключается в том, что с этим ничего нельзя сделать, кроме как дожидаться установления значений, а время установления является обратной экспоненциальной величиной разницы между начальным напряжением и метастабильным напряжением с фиксированной точкой. Мезохронная система обходит эту проблему, калибруя время выборки на границах тактового сигнала, чтобы получить наибольший запас по сравнению с метастабильным напряжением.

## Приложение D

# Коды операций

Реализация чужих спецификаций с моральной точки зрения эквивалентна переводу пятидесяти руководств пользователя видеоманитона с английского на японский язык.

—кролик

## Д.1 Общие примечания

Описания RTL операций опкода даны в блоковой форме, т.е. следующие строки кода

```
ПК ПК + 1 кк
ПК
ПК ПК + смещение
```

сохраняет значение начального  $PC + 1$  в  $q_0$  и значение начального  $PC + 1 + offset$  в  $PC$ .

Также следует отметить, что если операция  $PC$  не указана, подразумевается операция по умолчанию  $PC PC + 1$ , и что в результате приращения  $PC$  может быть выдано исключение, если  $PC$  входит в защищенную или недопустимую область кода.

## Д.2 Ленивые инструкции

Следующие инструкции могут потребовать несколько циклов для завершения выполнения и не останавливают счетчик программ (некоторые инструкции потребуют несколько циклов, но останавливают ПК до тех пор, пока они не будут завершены). Самое важное, что следует отметить, это то, что эти инструкции на самом деле не гарантируют, сколько времени потребуется для завершения. Две инструкции, запущенные в перекрывающемся порядке, могут завершиться не по порядку. Например, код

```
SPAWNC qn, метка1, q0
SPAWNC qn, метка2, q0
```

Может привести к тому, что возможность потока label1 будет возвращена после возможности потока label2. Если порядок возвращаемых значений имеет значение, рекомендуется использовать блокирующую промежуточную операцию перемещения очереди:

```
SPAWNC qn, метка1, q0
ПЕРЕМЕЩЕНИЕ q0, q1
SPAWNC qn, label2, q0
ПЕРЕМЕЩЕНИЕ q0, q1
```

Выполнение будет блокироваться каждый раз при выполнении инструкции MOVE q0, q1 до тех пор, пока q0 не получит значение.

Такое поведение многоцикловой инструкции называется «ленивым». Следующие инструкции

ленивый:

```
СПАУН
СПАУНК
РАСПРЕДЕЛИТЬ
АЛЛОКАТЕК
```

## Д.3 Краткое содержание инструкции

*Целочисленные арифметические инструкции:*

```
ДОБАВИТЬ qa, qb, qc
SUB qa, qb, qc
MUL qa, qb, qc
DIV qa, qb, qc
ADDC qa, n, qc
SUBC qa, n, qc
MULC qa, n, qc
DIVC qa, n, qc
```

*Логические инструкции оператора:*

```
И qa, qb, qc
ИЛИ qa, qb, qc
XOR qa, qb, qc
НЕ qa, qc
ANDC qa, n, qc
ORC qa, n, qc
XORC qa, n, qc
SHL кА, кБ, кК
SHR кА, кБ, кК
SRA кк, кб, кк
```



SHLC qa, n, qc  
SHRC k, n, k  
SRAC qa, n, qc

*Инструкции сравнения целых чисел:*

SEQ kA, kB, kK  
SNE kA, kB, kK  
SLT kA, kB, kK  
SGT kA, kB, kK  
CKB kA, kB, kK  
SGE kA, kB, kK  
SIC kK, kK  
SEQC k, n, k  
SNEC qa, n, qc  
SLTC qa, n, qc  
SGTC qa, n, qc  
SLEC qa, n, qc  
SGEC k, n, kK

*Преобразование чисел с плавающей точкой в целые числа:*

TOINT qa, qc  
TOREAL qa, qc

*Арифметические инструкции с плавающей точкой:*

FADD qa, qb, qc  
FSUB kK, kб, kK  
FMUL qa, qb, qc  
FDIV kK, kб, kK  
FADDC qa, n, qc  
FSUBC ka, n, kC  
FMULC qa, n, qc  
FDIVC qa, n, qc

*Инструкции сравнения чисел с плавающей точкой:*

FSEQ kK, kб, kK  
FSNE kK, kб, kK  
FSLT ka, kб, kK  
FSGT kK, kб, kK  
FSLE kA, kB, kK  
FSGE kK, kб, kK  
FSEQC qa, n, qc  
FSNEC kK, n, kK  
FSLTC ka, n, kC  
FSGTC k, n, kK  
FSLEC qa, n, qc  
FSGEC k, n, kK

*Инструкции по переходу и переходу:*

BR-лейбл  
BRL этикетка, qc BRZ qa, этикетка BRNZ qa, этикетка  
BRNE qa, этикетка  
BREL qa JMP qa

*Внутренние инструкции по обработке данных:*

ДВИГАТЬ qa, qc  
MOVECF n, qc  
MOVECL n, qc  
MOVECI n, qc  
MOVECS n, qc  
MOVECC n, qc  
УПАКОВКА qa, qb, qc, n  
РАСКН ка, кб, кк  
РАСКЛ кк, кб, кк  
ПАККИ qa, qb, qc  
РАСПАКОВАТЬ qa, qb, qc  
UNPACKC qa, n, qc  
EXTAG кк, кк  
SETTAG qa, qb, qc

*Инструкции по управлению очередью:*

FLUSHQ qc  
СПАУН qa, qb, qc  
SPAWNC контроль качества, этикетка, qc  
SPAWNЛ qa, qb, qc  
MAPQ кА, кБ, кК  
MAPQC qa, qb, qc  
MAPSQ qa, qb  
MAPDROP н  
UNMAPQ n  
ПОТРЕБЛЯЙТЕ qa  
EMPTY qa, qc EEQ qc

*Инструкции по управлению потоками и контекстом:*

PROCID кк  
LDCODE qa, qc OSIZE n

*Инструкции по запоминанию:*

PTRSIZE qa, qc  
РАСПРЕДЕЛИТЬ qa, qb, qc  
ALLOCATEC qa, n, qc  
MML qa, qb  
MMS qa, qb  
ОБМЕН qa, qb, qc  
PARCEL qa, qb, qc MSYNC

*Инструкции по обработке режимов и исключений:*

GETSTAT qc  
SETSTAT qa  
GETEX qc  
SETEX qa THROW

*Разные инструкции:*

СЛУЧАЙНЫЙ кк  
ПОДСКАЗКА т, подсказка

ДОБАВЛЯТЬ      кА, кБ, кК

#### Описание:

ADD (сложение) берет сумму qa и qb и возвращает результат в qc. qa и qb должны быть одного и того же целочисленного типа (word, packed int, packed short или packed char), в этом случае результат в qc будет иметь тот же тип, что и его предшественники. Кроме того, qa может быть возможностью, а qb может быть словом, в этом случае результатом будет возможность. Если qa или qb имеют несовместимые типы, qc будет помечен как недопустимый и возникнет исключение типа. Если qa является возможностью, а операция сложения со словом в qb не разрешена или приводит к недопустимой возможности, возникает исключение операции, а результат в qc является недопустимой возможностью.

Если qa является не копируемой возможностью, то успешная операция ADD исключает qa из очереди, даже если модификатор копирования/затирания для qa установлен на копирование и возникает исключение.

Операция ADD выполняется только в том случае, если доступны оба операнда qa и qb и нет противодействия на qc. В противном случае инструкция останавливается.

#### Операция:

```
если( тип(qa,qb) == слово )
    qc      кА + кБ
elif( тип(qa,qb) == упакованное целое )
    qc.a    qa.a + qb.a  qc.b
            к.а.б + к.б.б
elif( type(qa,qb) == (упакованный символ или упакованный короткий
символ) )
    qc.a    qa.a + qb.a
    qc.b    к.а.б      +
            к.б.б
    qc.c    qa.c + qb.c
    qc.d    к.а.д      +
            к.б.д
elif( (type(qa) == capabilities) && (type(qb) == word)
    ) temp qa + SEXT(qb & ADDRMASK) если (temp
    действителен) qc temp
    если (qa == не копируемый)
```

```
forceDequeue( qa ) // флаг ошибки, если бит копирования  
установлен на qa, иначе  
исключение операции throw
```

---

```
qc    неверный  
в противном случае  
выдать исключение  
типа
```

**Исключения:**

Исключение типа, исключение операции и исключение переполнения.

**Квалификации:**

Нет. Примечания:

Переполненные результаты также устанавливают соответствующий бит переполнения в поле типа qc.

АДДК            qa, n, qc

#### Описание:

ADDC (сложение с константой) берет сумму qa и n и возвращает результат в qc. qa может быть целочисленного типа (word, packed int, packed short или packed char), в этом случае результат в qc будет иметь тот же тип, что и его предшественники. В случае упакованных типов одна и та же константа добавляется к каждому подцелому числу. Кроме того, qa может быть возможностью, в этом случае результатом будет возможность. Если qa является возможностью, а операция сложения со словом в qb не разрешена или приводит к недопустимой возможности, возникает исключение операции, а результат в qc является недопустимой возможностью.

Если qa является не копируемой возможностью, то успешная операция ADDC исключает qa из очереди, даже если модификатор копирования/затирания для qa установлен на копирование и возникает исключение.

Операция ADDC выполняется только в том случае, если qa доступен и нет противодействия на qc. В противном случае инструкция останавливается.

#### Операция:

```
если( тип(qa) == слово )
контроль качества        qa + SEXT(n)
elif( тип(qa) == упакованное
      целое ) qc.a qa.a + n qc.b
      qa.b + n
elif( type(qa) == (упакованный символ или упакованный короткий) )
      qc.a     qa.a + n
      qc.b     qa.b + n
      qc.c     qa.c + n
      qc.d     к.а.д +
              н
elif( тип(qa) == возможность )
      temp qa + SEXT(n & ADDRMASK)
      если ( темп действителен ) qc
      temp
          если (qa == не копируемый)
              forceDequeue( qa ) // флаг ошибки, если бит копирования
              установлен на qa, иначе
```

исключение операции throw  
qc неверный

---

в противном случае  
выдать исключение  
типа

**Исключения:**

Исключение типа, исключение операции и исключение переполнения.

**Квалификации**

:

Никто.

**Примечания:**

Переполненные результаты также устанавливают соответствующий бит переполнения в поле типа qc.

**Описание:**

SUB (вычитание) берет разницу *qa* и *qb* и возвращает результат в *qc*. *qa* и *qb* должны быть одного и того же целочисленного типа (*word*, *packed int*, *packed short* или *packed char*), в этом случае результат в *qc* будет иметь тот же тип, что и его предшественники. Кроме того, *qa* может быть возможностью, а *qb* может быть словом, в этом случае результатом будет возможность. Если *qa* или *qb* имеют несовместимые типы, *qc* будет помечен как недопустимый и возникнет исключение типа. Если *qa* является возможностью, а операция сложения со словом в *qb* не разрешена или приводит к недопустимой возможности, возникает исключение операции, а результат в *qc* является недопустимой возможностью.

Если *qa* является не копируемой возможностью, то успешная операция SUB исключает *qa* из очереди, даже если модификатор копирования/затирания для *qa* установлен на копирование, и выдается исключение.

Операция SUB выполняется только в том случае, если доступны оба операнда *qa* и *qb* и нет противодействия на *qc*. В противном случае инструкция останавливается.

**Операция:**

```
если( тип(qa, qb) == слово )
    qc    qa - qb
elif( тип(qa, qb) == упакованное целое )
    qc.a  qa.a - qb.a
    qc.b  qa.b - qb.b
elif( type(qa, qb) == (упакованный символ или упакованный короткий
символ) )
    qc.a  qa.a - qb.a
    qc.b  qa.b - qb.b
    qc.c  qa.c - qb.c
    qc.d  qa.d - qb.d
elif( (type(qa) == capabilities) && (type(qb) == word)
    ) temp qa - SEXT(qb & ADDRMASK) если temp
действителен ) qc temp
    если (qa == не копируемый)
        forceDequeue(qa)
еще
```

исключение операции throw  
qc неверный

---

в противном случае  
выдать исключение  
типа

**Исключения:**

Исключение типа, исключение операции и исключение переполнения.

**Квалификации:**

Нет. Примечания:

Переполненные результаты также устанавливают соответствующий бит переполнения в поле типа qc.



## Описание:

СУБК            qa, n, qc

---

## Описание:

SUBC (вычитание с константой) берет разницу qa и n и возвращает результат в qc. qa может быть целочисленным типом (word, packed int, packed short или packed char), в этом случае результат в qc будет иметь тот же тип, что и его предшественники. В случае упакованных типов одна и та же константа вычитается из каждого подцелого числа. Кроме того, qa может быть возможностью, в этом случае результатом будет возможность. Если qa является возможностью, а операция сложения со словом в qb не разрешена или приводит к недопустимой возможности, возникает исключение операции, а результат в qc является недопустимой возможностью.

Если qa является не копируемой возможностью, то успешная операция SUBC исключает qa из очереди, даже если модификатор копирования/затирания для qa установлен на копирование и возникает исключение.

Операция SUBC выполняется только если qa доступен и нет противодействия на qc. В противном случае инструкция останавливается.

## Операция:

```
если( тип(qa) == слово )
контроль качества        qa - СЕКСТ(n)
elif( тип(qa) == упакованное
      целое ) qc.a qa.a - n qc.b
      qa.b - n
elif( type(qa) == (упакованный символ или
      упакованный короткий) ) qc.a qa.a - n qc.b
      qa.b - n qc.c    qa.c - n qc.d    к.а.д - н
elif( type(qa) == capacity ) temp
      qa - SEXT(n & ADDRMASK) if(
      temp is valid ) qc temp
      если (qa == не копируемый)
          forceDequeue( qa ) // флаг ошибки, если бит копирования
          установлен на qa, иначе
          исключение операции throw
          qc неверный
в противном случае
      выдать исключение
      типа
```

Исключения:

Исключение типа, исключение операции и исключение переполнения.

Квалификации

:

Нет.

Примечания:

Переполненные результаты также устанавливают соответствующий бит переполнения в поле типа `qc`.

МУЛ                    кА, кБ, кК

MUL (умножение) берет произведение `qa` и `qb` и возвращает младшие биты результата в `qc`. `qa` и `qb` должны быть одного и того же целочисленного типа (`word`, `packed int`, `packed short` или `packed char`), в этом случае результат в `qc` будет иметь тот же тип, что и его предшественники. Если `qa` или `qb` имеют несовместимые типы, `qc` будет помечен как недопустимый и будет вызвано исключение типа.

Операция MUL выполняется только в том случае, если доступны оба операнда `qa` и `qb` и нет противодействия на `qc`. В противном случае инструкция останавливается.

Операция:

```
если( тип(qa, qb) == слово ) qc  (qa * qb)
    & 0xFFFFFFFFFFFFFFFFFFFF
elif( type(qa, qb) == packed int ) qc.a
    (qa.a * qb.a) & 0xFFFFFFFF qc.b (qa.b
    * qb.b) & 0xFFFFFFFF
elif( type(qa, qb) == (упакованный символ или упакованный короткий
символ) )
    qc.a    (qa.a * qb.a) & 0xFFFF
    qc.b    (qa.b * qb.b) & 0xFFFF
    qc.c    (qa.c * qb.c) & 0xFFFF
    qc.d    (qa.d * qb.d) & 0xFFFF
в противном случае выдать
    исключение типа
```

Исключения:

Исключение типа и исключение переполнения.

Квалификации

:

Описание:

Нет.

Примечания:

---

Переполненные результаты также устанавливают соответствующий бит переполнения в поле типа qc.

МУЛЬК            qa, n, qc

---

Описание:

MULC (умножение с константой) берет произведение qa и n и возвращает младшие биты результата в qc. qa может быть целочисленного типа (word, packed int, packed short или packed char), в этом случае результат в qc будет иметь тот же тип, что и его предшественники. В случае упакованных типов одна и та же константа умножается на каждое подцелое число.

Операция MULC выполняется только если qa доступен и нет противодействия на qc. В противном случае инструкция останавливается.

Операция:

```
если( тип(qa) == слово ) qc      (qa * n)
    & 0xFFFFFFFFFFFFFFFF
elif( тип(qa) == упакованное целое )
    qc.a  (qa.a * n) & 0xFFFFFFFF qc.b
        (qa.b * n) & 0xFFFFFFFF
elif( type(qa) == (упакованный символ или упакованный короткий) )
    qc.a  (qa.a * n) & 0xFFFF
    qc.b  (qa.b * n) & 0xFFFF
    qc.c  (qa.c * n) & 0xFFFF
    qc.d  (qa.d * n) & 0xFFFF
в противном случае выдать
    исключение типа
```

Исключения:

Исключение типа и исключение переполнения.

Квалификации

:

Нет.

Примечания:

Переполненные результаты также устанавливают соответствующий бит переполнения в поле типа qc.

ДИВ

кА, кБ, кК

DIV (целочисленное деление) берет деление qa и qb и возвращает результат в qc. Нецелочисленные результаты усекаются. qa и qb должны быть одного и того же целочисленного типа (word, упакованный int, упакованный short или упакованный char), в этом случае результат в qc будет иметь тот же тип, что и его предшественники. Если qa или qb имеют несовместимые типы, qc будет помечен как недопустимый и возникнет исключение типа. Если делитель qb равен нулю, выдается исключение деления на ноль, и qc помечается как недопустимый, при этом конкретный упакованный компонент qc, который является ошибочным, помечается как переполненный.

Операция DIV выполняется только в том случае, если доступны оба операнда qa и qb и нет противодействия на qc. В противном случае инструкция останавливается.

Операция:

```
если( тип(qa,qb) == слово )
    если(qb == 0)
        выдать исключение типа деления на ноль
        (qc) недействительный, переполнение.а
    еще
контроль качества      кА / кБ
elif( тип(qa,qb) == упакованное целое )
    if(qb.a == 0)
        выдать исключение типа деления на ноль
        (qc.a)      недействительный, переполнение.а
    еще
        qc.a qa.a / qb.a если (qb.b
== 0)
        выдать исключение типа деления на ноль
        (qc.b)      недействительный, переполнение.б
    еще
        qc.b      qa.b / qb.b
elif( type(qa,qb) == (упакованный символ или упакованный
короткий) ) if(qb.a == 0)
    выдать исключение типа деления на ноль
    (qc.a)      недействительный, переполнение.а
    еще
```

#### Описание:

```
    qc.a qa.a / qb.a если (qb.b
== 0)
    выдать исключение типа деления на ноль


---


    (qc.b)    недействительный, переполнение.б
еще
    qc.b qa.b / qb.b если (qb.c
== 0)
    выдать исключение деления на ноль тип (qc.c) недопустимый,
    переполнение.c
еще
    qc.c qa.c / qb.c если (qb.d == 0)
    выдать исключение типа деления на ноль (qc.d) недопустимое,
    переполнение.d
еще
    qc.d    qa.d / qb.d
в противном случае выдать исключение типа
```

#### Исключения:

Исключение типа и исключение деления на ноль.

#### Квалификации:

Нет. Примечания: Нет.

ДИКВ

qa, n, qc

DIVC (деление с константой) берет деление qa и n и возвращает результат в qc. qa может быть целочисленного типа (word, packed int, packed short или packed char), в этом случае результат в qc будет иметь тот же тип, что и его предшественники. В случае упакованных типов одна и та же константа умножается на каждое подцелое число. Если делитель n равен нулю, выдается исключение деления на ноль, и qc помечается как недопустимый, при этом конкретный упакованный компонент qc, который является ошибочным, помечается как переполненный.

Операция DIVC выполняется только в том случае, если qa доступен и нет противодействия на qc. В противном случае инструкция останавливается.

Операция:

```
если( тип(qa) == слово )
    если(n == 0)
        выдать исключение типа деления на ноль
        (qc) недействительный, переполнение.а
    еще
        контроль качестваqa / SEXT(n)
elif( type(qa) == упакованное целое
    ) if(n == 0)
        выдать исключение типа деления на ноль
        (qc.a) недопустимый, тип overflow.a
        (qc.b) недействительный, переполнение.б
    еще
        qc.a qa.a / n qc.b
        qa.b / n
elif( type(qa) == (упакованный символ или упакованный
короткий) ) if(n == 0)
    выдать исключение типа деления на ноль
    (qc.a) недопустимый,
    переполнение.тип(qc.б) недопустимый,
    тип overflow.b(qc.c) недопустимый, тип
    overflow.c(qc.d) недействительный,
    переполнение.d
```

## Описание:

еще

$qс.а \ qа.а \ / \ n \ qс.б$

$qа.б \ / \ n \ qс.с$

---

$к.с. \ / \ н \ к.с.д$

$qа.д \ / \ n$

в противном случае выдать  
исключение

типа **Исключения:**

Исключение типа, исключение деления на ноль и исключение переполнения.

## Квалификации:

Нет. Примечания:

Переполненные результаты также устанавливают соответствующий бит переполнения в поле типа  $qс$ .

## И,ИЛИ,ИСКЛЮЧАЮЩЕЕ ИЛИкА, кБ, кК

AND, OR и XOR выполняют побитовые операции над qa и qb и возвращают результат в qc. qa и qb должны быть одного и того же целочисленного типа (word, packed int, packed short или packed char), в этом случае результат в qc будет иметь тот же тип, что и его предшественники. Если qa или qb имеют несовместимые типы, qc будет помечен как недопустимый и будет вызвано исключение типа.

Операция AND,OR,XOR выполняется только в том случае, если доступны оба операнда qa и qb и нет противодействия на qc. В противном случае инструкция останавливается.

### Операция:

```
OP — это одно из побитовых И, ИЛИ,  
XOR if( type(qa,qb) == word ) qc  
    qa OP qb  
elif( тип(qa,qb) == упакованное целое )  
    qc.a    qa.a OP qb.a qc.b  
    qa.b OP qb.b  
elif( type(qa,qb) == (упакованный символ или упакованный короткий  
символ) )  
    qc.a    qa.a OP qb.a  
    qc.b    qa.b OP qb.b  
    qc.c    qa.c OP qb.c  
    qc.d    qa.d OP qb.d  
в противном случае выдать  
исключение типа
```

### Исключения:

Тип исключения.

### Квалификации

:

Нет.

### Примечани

я: Нет.



## Описание:

---

НЕТ    контроль качества, контроль качества

NOT выполняет побитовую инверсию qa и возвращает результат в qc. qa должен быть целочисленного типа (word, packed int, packed short или packed char), в этом случае результат в qc будет иметь тот же тип, что и его предшественники. Если qa имеет несовместимый тип, qc будет помечен как недопустимый и будет вызвано исключение типа.

Операция НЕ выполняется только если qa доступен и нет противодействия на qc. В противном случае инструкция останавливается.

## Операция:

```
если( тип(qa) == слово
      ) qc  qa
elif( type(qa) == упакованное
      целое ) qc.a qa.a qc.b qa.b
elif( type(qa,qb) == (упакованный символ или упакованный короткий
символ) )
      qc.a      aa
      qc.b      qa.b
      qc.c      qa.c
      qc.d      qa.d
в противном случае
      выдать исключение
      типа
```

## Исключени

я:

Тип исключения.

## Квалифика

ции:

Описание:

Нет.

Примеч

ания:

Нет.

ИНДК,ОРК,ХОРК

ANDC, ORC и XORC выполняют побитовую операцию над qa и расширенным знаком n и возвращают результат в qc. qa может быть целочисленного типа (word, packed int, packed short или packed char), в этом случае результат в qc будет иметь тот же тип, что и его предшественники. В случае упакованных типов одна и та же константа применяется к каждому подцелому числу.

Операция ANDC,ORC,XORC выполняется только если qa доступен и нет противодействия на qc. В противном случае инструкция останавливается.

Операция:

```
OP — это одно из побитовых И,  
ИЛИ, XOR if( type(qa) == word )  
qc  qa OP SEXT(n)  
elif( тип(qa) == упакованное целое )  
    qc.a  qa.a OP n  
    qc.b  qa.b ОП n  
elif( type(qa) == (упакованный символ или упакованный короткий) )  
    qc.a  qa.a OP n  
    qc.b  qa.b ОП n  
    qc.c  qa.c OP n
```

**Описание:**

qc.d      qa.d OP n  
в противном случае  
выдать исключение  
типа

**Исключени**

**я:**

Тип исключения.

**Квалифика**  
**ции:**

Нет.

**Примеч**

**ания:**

Нет.

ШЛ

SHL (shift-left) выполняет логический сдвиг влево содержимого qa на количество цифр, указанное в qb, и возвращает результат в qc. Биты, сдвинутые влево, отбрасываются, а нули вставляются справа. qa и qb должны быть одного и того же целочисленного типа (word, packed int, packed short или packed char), в этом случае результат в qc будет иметь тот же тип, что и его предшественник. Если qa или qb имеют несовместимые типы, qc будет помечен как недопустимый и возникнет исключение типа.

**Описание:**

Операция SHL выполняется только в том случае, если доступны оба операнда qa и qb и нет противодействия на qc. В противном случае инструкция останавливается.

**Операция:**

```
если( тип(qa, qb) == слово
    ) qc  qa  (qb и 0x3F)
elif( type(qa, qb) == упакованное
    целое ) qc.a qa.a  (qb.a & 0x1F)
    qc.b  qa.b  (qb.b и 0x1F)
elif( type(qa, qb) == (упакованный символ или упакованный короткий
    символ) )
    qc.a  qa.a  (qb.a и 0xF)
    qc.b  qa.b  (qb.b и 0xF)
    qc.c  qa.c  (qb.c и 0xF)
    qc.d  qa.d  (qb.d и 0xF)
в противном случае
    выдать исключение
    типа
```

**Исключени**

**я:**

Тип исключения.

**Квалифика**  
**ции:**

Нет.

**Примеч**

**ания:**

Нет.

ШЛК

#### Описание:

SHLC (сдвиг влево на константу) выполняет логический сдвиг влево содержимого qa на количество цифр, указанное в n, и возвращает результат в qc. Биты, сдвинутые влево, отбрасываются, а нули вставляются справа. qa должен иметь целочисленный тип (word, packed int, packed short или packed char), в этом случае результат в qc будет иметь тот же тип, что и его предшественник. В случае, если qa является упакованным типом, каждое подслово будет сдвинуто влево на ту же величину. Если qa имеет несовместимый тип, qc будет помечен как недопустимый и возникнет исключение типа.

Операция SHLC выполняется только в том случае, если qa доступен и нет противодействия на qc. В противном случае инструкция останавливается.

#### Операция:

```
если( тип(qa) == слово )
контроль качества  qa (н и 0x3F)
elif( type(qa) == упакованное
      целое ) qc.a qa.a (н и 0x1F)
      qc.b  qa.b (н и 0x1F)
elif( type(qa) == (упакованный символ или упакованный короткий) )
      qc.a  qa.a (н и 0xF)
      qc.b  qa.b (н и 0xF)
      qc.c  qa.c (н и 0xF)
      qc.d  qa.d (н и 0xF)
в противном случае
      выдать исключение
      типа
```

#### Исключени

я:

Описание:

Тип исключения.

Квалифика  
ции:

Нет.

Примеч

ания:

Нет.

ШР

SHR (логический сдвиг вправо) выполняет логический сдвиг вправо содержимого qa на количество цифр, указанное в qb, и возвращает результат в qc. Биты, сдвинутые вправо, отбрасываются, а нули вставляются слева. qa и qb должны быть одного и того же целочисленного типа (word, packed int, packed short или packed char), в этом случае результат в qc будет иметь тот же тип, что и его предшественник. Если qa или qb имеют несовместимые типы, qc будет помечен как недопустимый и возникнет исключение типа.

Операция SHR выполняется только в том случае, если доступны оба операнда qa и qb и нет противодействия на qc. В противном случае инструкция останавливается.

Операция:

```
если( тип(qa, qb) == слово  
    ) qc  qa  (qb и 0x3F)
```

**Описание:**

```
elif( type(qa,qb) == упакованное
      целое ) qc.a qa.a (qb.a & 0x1F)
      qc.b  qa.b (qb.b и 0x1F)
elif( type(qa,qb) == (упакованный символ или упакованный короткий
символ) )
      qc.a  qa.a  (qb.a и 0xF)
      qc.b  qa.b  (qb.b и 0xF)
      qc.c  qa.c  (qb.c и 0xF)
      qc.d  qa.d  (qb.d и 0xF)
в противном случае
      выдать исключение
      типа
```

**Исключени**

**я:**

Тип исключения.

**Квалифика**

**ции:**

Нет.

**Примеч**

**ания:**

Нет.

ШРК

SHRC (логический сдвиг вправо на константу) выполняет логический сдвиг вправо содержимого qa на количество цифр, указанное в n, и возвращает результат в qc. Биты, сдвинутые вправо, отбрасываются, а нули вставляются слева. qa должен иметь целочисленный тип (word,

**Описание:**

packed int, packed short или packed char), в этом случае результат в qc будет иметь тот же тип, что и его предшественник. В случае, если qa является упакованным типом, каждое подслово будет сдвинуто влево на ту же величину. Если qa имеет несовместимый тип, qc будет помечен как недопустимый и возникнет исключение типа.

Операция SHRC выполняется только в том случае, если qa доступен и нет противодействия на qc. В противном случае инструкция останавливается.

**Операция:**

```
если( тип(qa) == слово )
контроль качества  qa (н и 0x3F)
elif( type(qa) == упакованное
      целое ) qc.a qa.a (н и 0x1F)
      qc.b  qa.b (н и 0x1F)
elif( type(qa) == (упакованный символ или упакованный короткий) )
      qc.a  qa.a (н и 0xF)
      qc.b  qa.b (н и 0xF)
      qc.c  qa.c (н и 0xF)
      qc.d  qa.d (н и 0xF)
в противном случае
      выдать исключение
      типа
```

**Исключени**

**я:**

Тип исключения.

**Квалифика**  
**ции:**



Описание:

Нет.

Примеч

ания:

Нет.

SRA

SRA (арифметический сдвиг вправо) выполняет арифметический (с сохранением знака) сдвиг вправо содержимого qa на количество цифр, указанное в qb, и возвращает результат в qc. Биты, сдвинутые вправо, отбрасываются, а значение бита знака сдвигается слева (ноль, если сдвигаемое число положительное, единица, если сдвигаемое число отрицательное). qa и qb должны быть одного и того же целочисленного типа (word, packed int, packed short или packed char), в этом случае результат в qc будет иметь тот же тип, что и его предшественник. Если qa или qb имеют несовместимые типы, qc будет помечен как недопустимый и возникнет исключение типа.

Операция SRA выполняется только в том случае, если доступны оба операнда qa и qb и нет противодействия на qc. В противном случае инструкция останавливается.

Операция:

```
если( тип(qa,qb) == слово )
контроль качестваqa SRA (qb и 0x3F)
elif( type(qa,qb) == упакованное
целое ) qc.a qa.a SRA (qb.a &
```

**Описание:**

```
0x1F) qc.b  qa.b SRA (qb.b &
0x1F)
elif( type(qa,qb) == (упакованный символ или упакованный короткий
символ) )
    qc.a  qa.a SRA (qb.a & 0xF)
    qc.b  qa.b SRA (qb.b & 0xF)
    qc.c  qa.c SRA (qb.c и 0xF)
    qc.d  qa.d SRA (qb.d и 0xF)
в противном случае
    выдать исключение
типа
```

**Исключени****я:**

Тип исключения.

**Квалифика****ции:**

Нет.

**Примеч****ания:**

Нет.

**SRAC**

SRAC (арифметический сдвиг вправо на константу) выполняет арифметический сдвиг вправо содержимого qa на количество цифр, указанное в n, и возвращает результат в qc. Биты, сдвинутые вправо, отбрасываются, а значение знакового бита сдвигается слева (ноль,

**Описание:**

если сдвигаемое число положительное, единица, если сдвигаемое число отрицательное). qa должен иметь целочисленный тип (word, packed int, packed short или packed char), в этом случае результат в qc будет иметь тот же тип, что и его предшественник. В случае, если qa является упакованным типом, каждое подслово будет сдвинуто влево на ту же величину. Если qa имеет несовместимый тип, qc будет помечен как недопустимый и возникнет исключение типа.

Операция SRAC выполняется только в том случае, если qa доступен и нет противодействия на qc. В противном случае инструкция останавливается.

**Операция:**

```
если( тип(qa) == слово )
контроль качестваqa SRA (n & 0x3F)
elif( type(qa) == упакованное
      целое ) qc.a qa.a SRA (n &
      0x1F) qc.b qa.b SRA (n &
      0x1F)
elif( type(qa) == (упакованный символ или упакованный короткий) )
      qc.a qa.a SRA (n & 0xF)
      qc.b qa.b SRA (n & 0xF)
      qc.c qa.c SRA (n & 0xF)
      qc.d qa.d SRA (n & 0xF)
в противном случае
      выдать исключение
      типа
```

**Исключени  
я:**

Описание:

Тип исключения.

Квалифика  
ции:

Нет.

Примеч

ания:

Нет.

SEQ, SLT, SLE

SEQ, SLT и SLE выполняют сравнение величин своих аргументов и выдают двоичный результат. SEQ проверяет, равны ли *qa* и *qb*; SLT проверяет, меньше ли *qa* *qb*; и SLE проверяет, меньше ли *qa* или равно *qb*. *qa* и *qb* должен быть того же целочисленного типа (*word*, *packed int*, *packed short* или *packed char*), в этом случае результат в *qc* будет иметь тот же тип, что и его предшественник. Если *qa* или *qb* имеют несовместимые типы, *qc* будет помечен как недопустимый и будет вызвано исключение типа.

Операция *Sxx* выполняется только в том случае, если доступны оба операнда *qa* и *qb* и нет противодействия на *qc*. В противном случае инструкция останавливается.

Операция:

OP — это один из арифметических операторов: `if( type(qa,qb) == word )`  
контроль качества `OP qb ? 1 : 0`

**Описание:**

```
elif( type(qa,qb) == packed int )
    qc.a qa.a OR qb.a ? 1 : 0 qc.b
    qa.b OR qb.b ? 1 : 0
elif( type(qa,qb) == (упакованный символ или упакованный короткий
символ) )
    qc.a    qa.a OR qb.a ? 1 : 0
    qc.b    qa.b OR qb.b ? 1 : 0
    qc.c    qa.c OR qb.c ? 1 : 0
    qc.d    qa.d OR qb.d ? 1 : 0
в противном случае
    выдать исключение
    типа
```

**Исключени**

**я:**

Тип исключения.

**Квалифика**

**ции:**

Нет.

**Примеч**

**ания:**

Нет.

**Описание:**

СИК    контроль качества, контроль качества

СИК проверяет, является ли qa возможностью. Если это так, тип слова 1 помещается в qc. В противном случае тип слова 0 помещается в qc.

Операция СИК выполняется только в том случае, если qa доступен и нет противодействия на qc. В противном случае инструкция останавливается.

**Операция:**

```
если ( тип(qa) == возможность
      ) qc   1
еще
контроль качества      0
```

**Исключени  
я:**

Никто.

**Квалифика  
ции:**

Нет.

**Примеч**

ания:

Нет.

SEQC,SLTC,SLEC

#### Описание:

SEQC, SLTC и SLEC выполняют сравнение величин своих аргументов и выдают двоичный результат. SEQC проверяет, равны ли qa и n; SLTC проверяет, меньше ли qa n; а SLEC проверяет, меньше ли qa или равно ли n. qa должен быть целочисленного типа (word, packed int, packed short или packed char), в этом случае результат в qc будет иметь тот же тип, что и его предшественник. Если qa имеет несовместимый тип, qc будет помечен как недопустимый и возникнет исключение типа.

Операция SxxC выполняется только в том случае, если qa доступен и нет противодействия на qc. В противном случае инструкция останавливается.

#### Операция:

OP — это один из арифметических

операторов: if( type(qa,qb) ==  
word ) qc qa OP n ? 1 : 0

elif( type(qa,qb) == packed int )

qc.a qa.a OP n ? 1 : 0 qc.b

qa.b OP n ? 1 : 0

elif( type(qa,qb) == (упакованный символ или упакованный короткий  
символ) )

qc.a qa.a OP n ? 1 : 0

qc.b qa.b OP n ? 1 : 0

qc.c qa.c OP n ? 1 : 0

qc.d qa.d OP n ? 1 : 0

в противном случае

выдать исключение

типа

#### Исключени

я:

Тип исключения.

Описание:

Квалифика  
ции:

Нет.

Примеч

ания:

Нет.

ТОИНТ контроль качества, контроль качества

ТОИНТ (преобразование числа с плавающей точкой в целое число) преобразует значение с плавающей точкой в `qa` в целое число, хранящееся в `qc`. Преобразование выполняется с использованием метода усечения или «округления до нуля», так что число 9,6 преобразуется в 9, а число -2,8 преобразуется в -2. Переполнение в любом знаковом экстремуме приводит к тому, что `qc` имеет максимальное целое число соответствующего знака, а бит переполнения устанавливается в поле типа `qc`. `qa` должен иметь тип с плавающей точкой, а результат в `qc` имеет тип `word`. Если `qa` имеет несовместимый тип, `qc` будет помечен как недопустимый и возникнет исключение типа.

Операция ТОИНТ выполняется только в том случае, если `qa` доступен и нет противодействия на `qc`. В противном случае инструкция останавливается.

Операция:



#### Описание:

---

```
если( тип(qa) == с плавающей
    точкой ) qc (слово) qa
    тип(qc) слово
в противном случае
    выдать исключение
    типа
```

#### Исключени

я:

Исключение типа и исключение переполнения.

#### Квалифика

ции:

Нет.

#### Примечания

:

Переполненные результаты также устанавливают соответствующий бит переполнения в поле типа qc. Попытка преобразовать + приведет к наибольшему положительному представимому целому числу в qc и установит бит переполнения qc. Аналогично, преобразование - приведет к наибольшему отрицательному представимому целому числу в qc и установит бит переполнения qc.

Попытка преобразовать NaN приведет к тому, что qc будет иметь недопустимый тип.

## Описание:

---

### TOREALКонтроль качества, контроль качества

TOREAL (преобразование целого числа в число с плавающей точкой) преобразует целочисленное значение в qa в ближайшее представимое значение с плавающей точкой, хранящееся в qc. qa должно иметь тип word, а результат в qc имеет тип с плавающей точкой. Если qa имеет несовместимый тип, qc будет помечен как недопустимый и будет вызвано исключение типа.

Операция TOREAL выполняется только в том случае, если qa доступен и нет противодействия на qc.

В противном случае выполнение инструкции зайдет в тупик.

## Операция:

```
если( тип(qa) == слово ) qc
    (с плавающей точкой) тип
    qa(qc) с плавающей точкой
в противном случае
    выдать исключение
    типа
```

Описание:

Исключени

я:

Тип исключения.

Квалифика

ции:

Нет.

Примеч

ания:

Нет.

ФАДД

FADD (сложение чисел с плавающей точкой) берет сумму qa и qb и возвращает результат в qc. qa и qb должны иметь тип с плавающей точкой, а результат qc должен иметь тип с плавающей точкой.

Операция FADD выполняется только в том случае, если доступны оба операнда qa и qb и нет противодействия на qc. В противном случае инструкция останавливается.

Описание:

Операция:

если( тип(qa, qb) == с плавающей  
точкой ) qc кА + кБ  
в противном случае  
выдать исключение  
типа

Исключени

я:

Исключение типа и исключение переполнения.

Квалифика

ции:

Нет.

Примечания

:

Переполненные результаты также устанавливают соответствующий бит переполнения в поле типа qc.

Если какой-либо операнд является NaN, результатом будет NaN.

FADDC

**Описание:**

FADDC (сложение чисел с плавающей точкой с константой) берет сумму `qa` и `n` и возвращает результат в `qc`. `qa` должен иметь тип с плавающей точкой, а результат `qc` должен иметь тип с плавающей точкой.

Операция FADDC выполняется только в том случае, если `qa` доступен и нет противодействия на `qc`. В противном случае инструкция останавливается.

**Операция:**

```
если( тип(qa) == с плавающей  
    точкой ) qc qa + n  
в противном случае  
    выдать исключение  
    типа
```

**Исключени**

**я:**

Исключение типа и исключение переполнения.

**Квалифика**

**ции:**

Описание:

Нет.

Примечания

:

Переполненные результаты также устанавливают соответствующий бит переполнения в поле типа qc.

Если qa — NaN, результат будет NaN.

ФСУБ

FSUB (вычитание чисел с плавающей точкой) берет разность qa и qb и возвращает результат в qc. qa и qb должны иметь тип с плавающей точкой, а результат qc должен иметь тип с плавающей точкой.

Операция FSUB выполняется только в том случае, если доступны оба операнда qa и qb и нет противодействия на qc. В противном случае инструкция останавливается.

Операция:

**Описание:**

если ( тип (qa, qb) == с плавающей  
точкой ) qc qa - qb  
в противном случае  
выдать исключение  
типа

**Исключени**

**я:**

Исключение типа и исключение переполнения.

**Квалифика**

**ции:**

Нет.

**Примечания**

:

Переполненные результаты также устанавливают соответствующий бит переполнения в поле типа qc.

Если какой-либо операнд является NaN, результатом будет NaN.

ФСУБК

**Описание:**

FSUBC (сложение чисел с плавающей точкой с константой) берет разность qa и n и возвращает результат в qc. qa должен иметь тип с плавающей точкой, а результат qc должен иметь тип с плавающей точкой.

Операция FSUBC выполняется только в том случае, если qa доступен и нет противодействия на qc. В противном случае инструкция останавливается.

**Операция:**

```
если( тип(qa) == с плавающей
    точкой ) qc qa - n
в противном случае
    выдать исключение
    типа
```

**Исключени**

**я:**

Исключение типа и исключение переполнения.

**Квалифика**

**ции:**



Описание:

Нет.

Примечания

:

Переполненные результаты также устанавливают соответствующий бит переполнения в поле типа `qc`.

Если `qa` — NaN, результат будет NaN.

ФМУЛЬ

FMUL (умножение чисел с плавающей точкой) берет произведение `qa` и `qb` и возвращает результат в `qc`. `qa` и `qb` должны иметь тип с плавающей точкой, а результат `qc` должен иметь тип с плавающей точкой.

Операция FMUL выполняется только в том случае, если доступны оба операнда `qa` и `qb` и нет противодействия на `qc`. В противном случае инструкция останавливается.

Операция:

**Описание:**

если( тип(qa, qb) == с плавающей  
точкой ) qc ка \* kb  
в противном случае  
выдать исключение  
типа

**Исключени  
я:**

Исключение типа и исключение переполнения.

**Квалифика  
ции:**

Нет.

**Примечания**

:

Переполненные результаты также устанавливают соответствующий бит переполнения в поле типа qc.

Если какой-либо операнд является NaN, результатом будет NaN.

FMULC

**Описание:**

FMULC (умножение чисел с плавающей точкой на константу) берет произведение qa и n и возвращает результат в qc. qa должен иметь тип с плавающей точкой, а результат qc должен иметь тип с плавающей точкой.

Операция FMULC выполняется только в том случае, если qa доступен и нет противодействия на qc. В противном случае инструкция останавливается.

**Операция:**

```
если( тип(qa) == с плавающей  
    точкой ) qc = qa * n  
в противном случае  
    выдать исключение  
    типа
```

**Исключени**

**я:**

Исключение типа и исключение переполнения.

**Квалифика**

**ции:**

Описание:

Нет.

Примечания

:

Переполненные результаты также устанавливают соответствующий бит переполнения в поле типа qc.

Если qa — NaN, результат будет NaN.

ФДИВ

FDIV (деление с плавающей точкой) делит qa на qb и возвращает частное в qc. qa и qb должны быть типа с плавающей точкой, а результат qc — типа с плавающей точкой. Если qb равен нулю, выдается исключение деления на ноль, а результат qc помечается как недопустимый.

Операция FDIV выполняется только в том случае, если доступны оба операнда qa и qb и нет противодействия на qc. В противном случае инструкция останавливается.

кА, кБ, кК

---

Описание:

Операция:

если( тип(qa, qb) == с плавающей  
точкой ) qc кА / кБ  
в противном случае  
выдать исключение  
типа

Исключени

я:

Исключение типа, исключение переполнения и исключение деления на ноль.

Квалифика

ции:

Нет.

Примечания

:

Переполненные результаты также устанавливают соответствующий бит переполнения в поле типа qc.

Если какой-либо операнд является NaN, результатом будет NaN.

FDIVC

**Описание:**

FDIVC (деление с плавающей точкой на константу) делит qa на n и возвращает результат в qc. qa должен быть типом с плавающей точкой, а результат qc — типом с плавающей точкой. Если n равно нулю, выдается исключение деления на ноль, а результат qc помечается как недопустимый.

Операция FDIVC выполняется только в том случае, если qa доступен и нет противодействия на qc. В противном случае инструкция останавливается.

**Операция:**

```
если( тип(qa) == с плавающей
    точкой ) qc qa / n
в противном случае
    выдать исключение
    типа
```

**Исключени  
я:**

Исключение типа, исключение переполнения и исключение деления на ноль.

кА, кБ, кК

---

Описание:

Квалифика

ции:

Нет.

Примечания

:

Переполненные результаты также устанавливают соответствующий бит переполнения в поле типа qc.

Если qa — NaN, результат будет NaN.

---

**Описание:**

FSEQ,FSLT,FSLE      кА, кБ, кК

FSEQ, FSLT и FSLE выполняют сравнение величин своих аргументов и выдают двоичный целочисленный результат. FSEQ проверяет, равны ли qa и qb; FSLT проверяет, меньше ли qa qb; а FSLE проверяет, меньше ли qa или равно qb qa, и qb должен быть типом с плавающей точкой. Результат qc имеет тип word. Если qa или qb имеют несовместимые типы, qc будет помечен как недопустимый и будет вызвано исключение типа.

Операция FSxx выполняется только в том случае, если доступны оба операнда qa и qb и нет противодействия на qc. В противном случае инструкция останавливается.

**Операция:**

OP — это одна из арифметических операций:

если( тип(qa, qb) == с плавающей  
точкой ) qc qa OP qb ? 1 : 0  
тип(кк)      слово

в противном случае  
выдать исключение  
типа

**Исключени**

**я:**



---

Описание:

Тип исключения.

Квалифика  
ции:

Нет.

Примечания

:

Если какой-либо из операндов является NaN, результат помечается как недействительный.  
FSEQC,FSLTC,FSLEC ка, kb, кс

FSEQC, FSLTC и FSLEC выполняют сравнение величин своих аргументов и выдают двоичный результат. FSEQC проверяет, равны ли qa и n; FSLTC проверяет, меньше ли qa n; и FSLEC проверяет, меньше ли qa или равно ли n. qa должен иметь тип с плавающей точкой, а результат в qc имеет тип word. Если qa имеет несовместимый тип, qc будет помечен как недопустимый и будет вызвано исключение типа.

Операция FSxxC выполняется только в том случае, если qa доступен и нет противодействия на qc. В противном случае инструкция останавливается.

Операция:

OP — это одна из арифметических операций:

если( тип(qa,qb) == с плавающей  
точкой ) qc qa OP n ? 1 : 0

---

**Описание:**

тип (КК) слово  
в противном случае  
выдать исключение  
типа

**Исключени**

я:

Тип исключения.

**Квалифика**

ции:

Нет.

**Примечания**

:

Если  $qa$  — NaN, результат помечается как недействительный.  
БР компенсировать

BR (безусловный переход) добавляет число, указанное в поле смещения, к увеличенному счетчику программ. Выполнение немедленно начинается с нового значения РС; слоты задержки перехода отсутствуют.

**Операция:**

ПК ПК + 1

---

Описание:

ПК      ПК + смещение

Исключени

я:

Если место назначения ПК находится на защищенной или недопустимой странице, выдается исключение.

Квалифика  
ции:

Нет.

Примеч

ания:

Нет.

---

**Описание:****БРИЛ****смещение, кк**

BRL (безусловный переход со ссылкой) добавляет число, указанное в поле смещения, к увеличенному счетчику программ. Выполнение немедленно начинается с нового значения РС; слотов задержки перехода нет. Увеличенное смещение счетчика программ относительно начала кода (будь то метод, объект или абсолютная ссылка) хранится в qc как тип данных word; выполнение останавливается, если qc заполнен и применяется обратное давление.

Операция BRL выполняется только в том случае, если нет противодействия на qc. В противном случае инструкция останавливается.

**Операция:**
$$\text{ПК} \quad \text{ПК} + 1$$
$$\text{кк} \quad \text{ПК}$$
$$\text{ПК} \quad \text{ПК} + \text{смещение}$$
**Исключени****я:**

Если место назначения ПК находится на защищенной или недопустимой странице, выдается исключение.

**Квалифика****ции:**

---

Описание:

Нет.

Примеч

ания:

Нет.

БРЗ                      qa, смещение, подсказка

BRZ (переход, если ноль) добавляет число, указанное в поле смещения, к увеличенному счетчику программ, если значение в qa равно нулю; в противном случае счетчик программ просто увеличивается до следующей инструкции. qa должен иметь тип word. Выполнение немедленно начинается с нового значения PC; слотов задержки перехода нет.

Операция BRZ выполняется только в том случае, если qa доступен и нет противодействия на qc. В противном случае инструкция останавливается.

Операция:

```
если( тип(qa) == слово )
    если (qa == 0)
        ПК    ПК + 1 + смещение
    еще
        ПК    ПК + 1
```

---

Описание:

Исключени

я:

Если место назначения ПК находится на защищенной или недопустимой странице, выбрасывается исключение. Исключение типа выбрасывается, если тип `qa` не является `word`.

Квалифика

ции:

Нет.

Примечания

:

Поле подсказки — это 8-битное число, зависящее от реализации и служащее подсказкой для предсказания ветвления. Семантика подсказки такова, что неправильная подсказка для ветвления все равно приводит к правильному, но более медленному выполнению.

Фактическому значению подсказки разрешено иметь несогласованную с кэшем мутацию во время выполнения, как считает нужным динамический аппаратный предсказатель ветвления.

БРНЗ                      `qa`, смещение, подсказка

БРНЗ (переход, если не ноль) добавляет число, указанное в поле смещения, к увеличенному счетчику программ, если значение в `qa` не равно нулю; в противном случае счетчик программ просто увеличивается до следующей инструкции. `qa` должен иметь тип `word`.

---

**Описание:**

Выполнение немедленно начинается с нового значения РС; слотов задержки перехода нет.

Операция BRNZ выполняется только в том случае, если qa доступен и нет противодействия на qs. В противном случае инструкция останавливается.

**Операция:**

```
если ( тип (qa) == слово )
    если (qa != 0 )
        ПК    ПК + 1+ смещение
    еще
        ПК    ПК + 1
```

**Исключени**

**я:**

Если место назначения ПК находится на защищенной или недопустимой странице, выбрасывается исключение. Исключение типа выбрасывается, если тип qa не является word.

**Квалифика**

**ции:**

Нет.

**Примечания**

:

Поле подсказки — это 8-битное число, зависящее от реализации и служащее подсказкой для предсказания ветвления. Семантика подсказки такова, что неправильная подсказка для ветвления все равно приводит к правильному, но более медленному выполнению.

---

**Описание:**

Фактическому значению подсказки разрешено иметь несогласованную с кэшем мутацию во время выполнения, как считает нужным динамический аппаратный предсказатель ветвления.

БРНЕ                      qa, смещение

БРНЕ (переход, если не пустой) добавляет число, указанное в поле смещения, к увеличенному счетчику программ, если qa не пустой; в противном случае счетчик программ просто увеличивается до следующей инструкции. Данные в qa не затрагиваются этой инструкцией. Выполнение немедленно начинается с нового значения РС; слотов задержки перехода нет.

**Операция:**

если (qa != пусто)

ПК    ПК + 1 + смещение

еще

ПК    ПК + 1

**Исключени**

**я:**

Если место назначения ПК находится на защищенной или недопустимой странице, выдается исключение.



---

Описание:

Квалифика

ции:

Квалификатор игнорируется этой инструкцией; qa никогда не удаляется из очереди.

Примечани

я: нет.

БРЭЛ

qa

BREL (безусловный относительный переход) добавляет число в qa к увеличенному счетчику программ. qa должен иметь тип word. Выполнение немедленно начинается с нового значения PC; слотов задержки перехода нет.

Операция BREL выполняется только в том случае, если qa доступен и нет противодействия на qc. В противном случае инструкция останавливается.

Операция:

если( тип(qa) == слово )

ПК      ПК + 1 + qa

Исключени

я:

Если место назначения ПК находится на защищенной или недопустимой странице, выбрасывается исключение. Исключение типа выбрасывается, если тип qa не является word.

---

Описание:

Квалифика

ции:

Нет.

Примеч

ания:

Нет.

СПМ

qa, намек

JMP (безусловный переход) устанавливает значение в PC в значение в qa. Выполнение немедленно начинается с нового значения PC; слотов задержки перехода нет. qa должен иметь тип word.

Операция JMP выполняется только в том случае, если qa доступен и нет противодействия на qc. В противном случае инструкция останавливается.

Операция:

если( тип(qa) == слово )

ПК qa

Исключени

я:

Если место назначения ПК находится на защищенной или недопустимой странице, выдается исключение.

---

Описание:

Квалифика

ции:

Нет.

Примечания

:

Поле подсказки — это специфичное для реализации 48-битное число, которое служит подсказкой назначения прогнозирования перехода. Семантика подсказки такова, что неправильная подсказка перехода все равно приводит к правильному, но более медленному выполнению. Фактическому значению подсказки разрешено иметь несогласованную с кэшем мутацию во время выполнения, как считает нужным динамический аппаратный предсказатель перехода.

ДВИГАТЬСЯ контроль качества, контроль качества

MOVE (перемещение) берет значение в qa и помещает его в qc. Точное состояние очередей после инструкции MOVE зависит от модификаторов @ (копировать/затирать), примененных к спецификаторам очередей.

Операция MOVE выполняется только в том случае, если qa доступен и нет противодействия на qc. В противном случае инструкция останавливается.

Операция:

контроль качества      qa

---

Описание:

Исключени

я:

Исключение операции возникает, если оператор копирования применяется к данным в qa, помеченным как не копируемые. Результат в qc помечается как недействительный, а исходное значение остается нетронутым в qa.

Квалифика

ции:

Нет.

Примечания

:

Точная семантика варьируется в зависимости от использования модификатора @.

**Описание:****MOVECF**

MOVECF (перемещение константы с плавающей точкой) берет 32-битную константу с плавающей точкой, указанную в n, преобразует ее в ближайшее 64-битное число с плавающей точкой ADAM и помещает правильно типизированный результат в qc. Точное состояние qc после инструкции MOVECF зависит от модификатора @ (копировать/затирать), примененного к спецификатору очереди.

Операция MOVECF выполняется только в том случае, если нет противодействия на qc. В противном случае инструкция останавливается.

**Операция:**

qc (плавающая точка) n тип(qc)  
плавающая точка

**Исключени**

я:

Никто.

Описание:

Квалифика  
ции:

Нет.

Примечания

:

Из-за преобразования из 32-битного представления, хранящегося в коде операции, в 64-битное стандартное представление с плавающей точкой ADAM результат в qc может демонстрировать некоторую небольшую ошибку округления по сравнению с желаемой константой.

Точная семантика варьируется в зависимости от использования модификатора @.

**MOVECL**

MOVECL (перемещение длинной целой константы) берет 32-битную константу, указанную в n, расширяет ее до собственного 64-битного слова ADAM и помещает правильно типизированный результат в qc. Точное состояние qc после инструкции MOVECL зависит от модификатора @ (копировать/затирать), примененного к спецификатору очереди.

**Описание:**

Операция MOVECL выполняется только в том случае, если нет противодействия на qс. В противном случае инструкция останавливается.

**Операция:**

qс SEXT (n) тип (qс)  
слово

**Исключени**

**я:**

Никто.

**Квалифика**

**ции:**

Нет.

**Примечания**

:

Точная семантика варьируется в зависимости от использования модификатора @.  
MOVECI

**Описание:**

MOVECI (перемещение упакованной целочисленной константы) берет 32-битную константу, указанную в n, помещает ее в нижние биты упакованного целого числа, устанавливает верхние биты упакованного целого числа в ноль и помещает правильно типизированный результат в qc. Точное состояние qc после инструкции MOVECI зависит от модификатора @ (копировать/затирать), примененного к спецификатору очереди.

Операция MOVECI выполняется только в том случае, если нет противодействия на qc. В противном случае инструкция останавливается.

**Операция:**

qc.a 0 qc.b

n

тип (KK) упакованное целое число

**Исключени**

я:

Никто.



Описание:

Квалифика

ции:

Нет.

Примечания

:

Точная семантика варьируется в зависимости от использования модификатора @.

MOVECS

MOVECS (перемещение упакованной короткой константы) берет двойную 16-битную упакованную короткую константу, указанную в *n*, помещает ее в нижние биты упакованной короткой константы, устанавливает верхние биты упакованной короткой константы в ноль и помещает правильно типизированный результат в *qs*. Точное состояние *qs* после инструкции MOVECS зависит от модификатора @ (копировать/затирать), примененного к спецификатору очереди.

Операция MOVECS выполняется только в том случае, если нет противодействия на *qs*. В противном случае инструкция останавливается.

Описание:

Операция:

qс.а 0

qс.б 0

qс.с н[31:16]

qс.д н[15:0]

тип (кк) упакованный короткий

Исключени

я:

Никто.

Квалифика

ции:

Нет.

Примечания

:

Точная семантика варьируется в зависимости от использования модификатора @.  
MOVECC

**Описание:**

MOVECC (переместить упакованную константу символа unicode) берет двойную 16-битную упакованную константу символа unicode, указанную в n, помещает ее в нижние биты упакованного символа, устанавливает верхние биты упакованного символа в ноль и помещает правильно типизированный результат в qc. Точное состояние qc после инструкции MOVECC зависит от модификатора @ (копировать/затирать), примененного к спецификатору очереди.

Операция MOVECC выполняется только в том случае, если нет противодействия на qc. В противном случае инструкция останавливается.

**Операция:**

qc.a	0
qc.b	0
qc.c	n[31:16]
qc.d	n[15:0]
тип (kk)	упакованный характер

**Исключени**

**я:**

Никто.

Описание:  
Квалифика  
ции:  
Нет.

Примечания

:

Точная семантика варьируется в зависимости от использования модификатора @.

---

Описание:

PACKN (Pack Anything) берет данные в qa и вставляет их в позицию, указанную n, в данные из qb, и помещает результат в qc. qa должен иметь тип word, а qb должен иметь упакованный целочисленный тип. Результат в qc имеет тот же тип, что и qb.

Операция PACKN выполняется только в том случае, если доступны оба операнда qa и qb и нет противодействия на qc. В противном случае инструкция останавливается.

## Операция:

```

если( тип(qa) == слово )
    если( тип(qb) == упакованное целое )
        если( н == 0 )
            qc.a    кА и 0xFFFFFFFF
            qc.b    qb.b
        еще
            qc.a    qb.a
            qc.b    кА и 0xFFFFFFFF
    elif( type(qb) == упакованный short или упакованный char )
        если( н == 0 )
            qc.a    кА и 0xFFFF
            qc.b    qb.b
            qc.c    qb.c
            qc.d    qb.d
        elif( н == 1 )
            qc.a    qb.a
            qc.b    кА и 0xFFFF
            qc.c    qb.c
            qc.d    qb.d
        elif( н == 2 )
            qc.a    qb.a
            qc.b    qb.b
            qc.c    кА и 0xFFFF
            qc.d    qb.d
        еще
            qc.a    qb.a
            qc.b    qb.b
            qc.c    qb.c
            qc.d    кА и 0xFFFF
    в противном случае выдать
        исключение типа
еще

```

кА, кБ, кК

Описание:

исключение типа throw

Исключения:

---

Тип исключения.

Квалификации:

Нет. Примечания: Нет.

## ПАКХ

РАСКН (Pack High Half of Packed Short or Char) берет упакованные целочисленные данные в `qa`, маскирует данные и вставляет их в верхнюю половину `qb`, а результат помещает в `qc`. `qb` должен иметь тип упакованный short или упакованный char. Результат в `qc` имеет тот же тип, что и `qb`.

Операция РАСКН выполняется только в том случае, если доступны оба операнда `qa` и `qb` и нет противодействия на `qc`. В противном случае инструкция останавливается.

### Операция:

если (тип (`qa`) == упакованный int && тип (`qb`) == упакованный short или упакованный char)

```
qc.a    qa.a и
0xFFFF qc.b    qa.b
&0xFFFF qc.c    qb.c
qc.d    qb.d
```

в противном случае  
выдать исключение  
типа

### Исключения:

Тип исключения.

### Квалификации

:

Нет.

### Примечания

: Нет.

**Описание:****ПАКЛ**

PACKL (Pack Low Half of Packed Short or Char) берет упакованные целочисленные данные в qa, маскирует данные и вставляет их в нижнюю половину qb, а результат помещает в qc. qb должен иметь тип упакованный short или упакованный char. Результат в qc имеет тот же тип, что и qb.

Операция PACKL выполняется только в том случае, если доступны оба операнда qa и qb и нет противодействия на qc. В противном случае инструкция останавливается.

**Операция:**

если (тип (qa) == упакованный int && тип (qb) == упакованный short или упакованный char)

```
qc.a    qb.a qc.b
      qb.b qc.c qa.a &
0xFFFF qc.d  qa.b и
0xFFFF
```

в противном случае  
выдать исключение  
типа

**Исключения:**

Тип исключения.

**Квалификации**

:



Описание:

Нет.

Примечания

: Нет.

### ПАККИ

РАСКИ (Pack to Packed Integer – упаковать в упакованное целое число) берет данные слов в qa и qb, маскирует данные и упаковывает их в упакованное целое число, хранящееся в qc.

Операция РАСКИ выполняется только в том случае, если доступны оба операнда qa и qb и нет противодействия на qc. В противном случае инструкция останавливается.

Операция:

```
если( тип(qa, qb) == слово )
qc.a qa & 0xFFFFFFFF qc.b qb и
0xFFFFFFFF тип(qc)
    упакованное целое число,
иначе тип выбросаисключение
```

Исключения:

Тип исключения.

Квалификации

:

Нет.

Примечания

: Нет.

### РАСПАКОВАТЬ

**Описание:**

UNPACK (Распаковка) берет упакованный целочисленный тип qa, извлекает и расширяет данные в месте qb в qc. Результат qc имеет тип word.

Операция UNPACK выполняется только в том случае, если доступны оба операнда qa и qb и нет противодействия на qc. В противном случае инструкция останавливается.

**Операция:**

```
если (тип(qb) == слово)
    если (тип(qa) == упакованное целое)
        если (qb == 0)
            контроль качестваSEXT(qa.a)
            еще
            контроль качестваSEXT(qa.b)
        elif(тип(qa) == упакованный short или упакованный char)
            если (qb == 0) qc
                SEXT(qa.a)
            elif(qb == 1) qc
                SEXT(qa.b)
            elif(qb == 2)
                контроль качестваSEXT(qa.c)
                еще
                контроль качестваSEXT(qa.d)
                в противном случае выдать
                исключение типа
            в противном случае выдать
            исключение типа
```

**Исключения:**

Тип исключения.

**Квалификации**

:

кА, кБ, кК

---

Описание:

Нет.

Примечания

: Нет.

---

**Описание:**

UNPACKC                      qa, n, qc

UNPACKC (Распаковка с константой) берет упакованный целочисленный тип qa, извлекает и расширяет данные в позиции n в qc. Результат qc имеет тип word.

Операция UNPACKC выполняется только в том случае, если qa доступен и нет противодействия на qc.

В противном случае выполнение инструкции зайдет в тупик.

**Операция:**

```
если (тип (qa) == упакованное целое)
    если (n == 0)
        контроль качества      SEXT(qa.a)
        еще
        контроль качества      SEXT(qa.b)
    elif(тип(qa) == упакованный short или упакованный char)
        если (n == 0) qc
            SEXT(qa.a)
        elif(n == 1) qc
            SEXT(qa.b)
        elif(n == 2)
            контроль качества      SEXT(qa.c)
            еще
            контроль качества      SEXT(qa.d)
    в противном случае выдать
    исключение типа
```

**Исключения:**

Тип исключения.

**Квалификации:**

Нет. Примечания:

Нет.

FLUSHQ                      контроль качества

---

**Описание:**

FLUSHQ (Flush Queue) — это инструкция специального формата, где `qc` интерпретируется как непосредственная константа. FLUSHQ отбрасывает все значения, находящиеся в очереди, заданные немедленной константой `qc`. Функция FLUSHQ для очереди, которая имеет сопоставления с другими контекстами, будь то сопоставления головы или хвоста, НЕПРЕДСКАЗУЕМА. Если `qc` уже пуст, ничего не происходит и выполнение продолжается.

**Операция:**

контроль качества      пустой

**Исключения:**

Выдает исключение сопоставления, если у `qc` есть какие-либо сопоставления.

**Квалификации:**

Нет. Примечания:

Нет.

Описание: \_\_\_\_\_  
ПРОЦИД            контроль качества

PROCID (Get Process ID) помещает значение текущего идентификатора контекста в qc. qc — это возможность с установленным битом владельца. Кроме того, устанавливаются биты чтения и записи. Если идентификатор контекста должен быть передан другому потоку, необходимо позаботиться о правильной установке разрешений.

Операция PROCID выполняется только в том случае, если нет противодействия на qc. В противном случае инструкция останавливается.

Операция:  
контроль качества            идентификатор контекста

Исключения:  
Никто.

Квалификации:  
Нет. Примечания:

Нет.

## PTRSIZE контроль качества, контроль качества

PTRSIZE (Get Pointer Size) вычисляет размер области данных, на которую указывает возможность в qa, и помещает размер в словах в qc. Операция PTRSIZE действительна для любой возможности, независимо от ее разрешений. Результат в qc имеет тип слова.

Операция PTRSIZE выполняется только в том случае, если qa доступен и нет противодействия на qc.

В противном случае выполнение инструкции зайдет в тупик.

## Операция:

```
если( тип(qa) == возможность )
    контроль качества sizeof(qa) в словах
    type(qc) слово
в противном случае выдать
    исключение типа
```

## Исключения:

Тип исключения.

## Квалификации:

Нет. Примечания:

Нет.

ПОТРЕБЛЯТЬ            qa

CONSUME (Consume Data) считывает ровно один фрагмент данных из qa и отбрасывает его. Если qa изначально пуст, CONSUME блокируется.

## Операция:

```
пока (qa пуст) задержка
если (нет оператора @ в qa) вывести из
    очереди заголовков qa
```

Описание:

Исключения:

Никто.

Квалификации:

Нет. Примечания:

Нет.



SEMPY (Set if Empty) – это специальная инструкция формата, где qa интерпретируется как непосредственная константа. SEMPY проверяет, пуста ли очередь, заданная непосредственно константой qa, и если да, то помещает целое число 1 в qc. В противном случае в qc записывается 0. Тип результата qc – word.

**Операция:**

если ((qa & 0x7F) пусто) qc 1  
еще  
контроль качества 0  
тип (кк) слово

**Исключения:**

Никто.

**Квалификации:**

Нет. Примечания:

Нет.

ЭЭ qa

EEQ (force Empty Queue) – это специальная инструкция формата, где qa интерпретируется как непосредственная константа. EEQ проверяет, пуста ли очередь, указанная непосредственная константа qa, и если да, то увеличивает РС; если нет, РС остается постоянным, а планировщику сообщается о приводящем останове.

**Операция:**

если ((qa & 0x7F) пусто) ПК ПК + 1  
еще  
ПК ПК

Описание:

Исключения:

Никто.

Квалификации:

Нет. Примечания:

Эта инструкция усложняет реализацию ядра процессора. Альтернативой было бы использование SEMPTY и инструкции BRZ для создания программного цикла проверки пустоты очереди. Однако в целях обратной совместимости со старой ISA она включена в документацию.

СЛУЧАЙНЫЙ контроль качества

RANDOM (Generate Random Number) помещает криптографически безопасное случайное целое число типа word в qc. RANDOM может быть реализован как внешнее аппаратное устройство для процессора. Поскольку для каждой инструкции RANDOM необходимо собрать 64 бита энтропии, можно запрашивать случайные числа быстрее, чем процессор или устройство способны их генерировать. В этом случае операция блокируется до тех пор, пока случайное число не станет доступным. Чтобы сгладить шаблоны спроса, устройство генерации чисел может выбрать постановку в очередь нескольких заранее сгенерированных чисел.

Операция RANDOM выполняется только в том случае, если нет противодействия на qc. В противном случае инструкция останавливается.

Операция:

qc случайное число между и типом (qc) слово

Исключения:

Никто.

Квалификации:

Нет. Примечания:

Точная реализация функции RANDOM должна быть раскрыта публично, прежде чем ей можно будет доверять. Более подробную информацию о криптографически безопасных случайных числах можно найти в Приложении D.6 «Генерация случайных чисел» стандарта IEEE 1363-2000 и в RFC1750, «Рекомендации по случайности для безопасности». Пользователь, желающий проверить свойства случайности инструкции RANDOM, может обратиться к «Универсальному статистическому тесту для генераторов случайных

---

Описание:

битов» Ули М. Маурера, Институт теоретической информатики, ETH Zurich", 1992, Журнал криптологии, т. 5, № 2.

ПОЛУЧИТЬСТАТ контроль качества

GETSTAT (Get Status Register) копирует содержимое регистра статуса в qc. Некоторые части регистра статуса зависят от реализации. qc имеет тип word.

Операция GETSTAT выполняется только в том случае, если нет противодействия на qc. В противном случае инструкция останавливается.

Операция:

тип регистра статуса qc(qc) слово

Исключения:

Никто.

Квалификации:

Нет. Примечания:

Значение битов регистра состояния см. в примечаниях к реализации и спецификации архитектуры.

SETSTAT                      qa

SETSTAT (Set Status Register) копирует содержимое qa в изменяемые части регистра статуса. Некоторые части регистра статуса зависят от реализации. qa должен иметь тип word.

Операция SETSTAT выполняется только в том случае, если нет противодействия на qc. В противном случае инструкция останавливается.

Операция:

если(тип(qa) == слово)

    регистр статуса      qa

в противном случае выдать  
    исключение типа

Описание: \_\_\_\_\_  
Исключения:

Тип исключения.

Квалификации:

Нет. Примечания:

Пожалуйста, обратитесь к примечаниям по реализации и спецификации архитектуры для получения информации о значении битов регистра состояния. Некоторые биты регистра состояния доступны только для чтения и не затрагиваются SETSTAT.

ГЕТЕКС                      контроль качества

GETEX (Get Exception Context ID) помещает текущий контекстный ID обработчика исключений в qc. Разрешения на ID обработчика исключений устанавливаются на непрозрачный и владелец.

Операция GETEX выполняется только в том случае, если нет противодействия на qc. В противном случае инструкция останавливается.

Операция:

qc Исключение Регистр Тип(qc) Возможности  
Разрешения(qc) Непрозрачный, Владелец

Исключения:

Никто.

Квалификации:

Нет. Примечания:

Нет.

СЕТЕКС                      qa

SETEX (Set Exception Context ID) устанавливает идентификатор обработчика исключений текущего контекста в качестве возможности в qa. Операция блокируется, если qa применяет обратное давление.

---

Описание:

**Операция:**

если (тип (qa) == возможность)  
Регистр исключений qa else выдать  
исключение типа

**Исключения:**

Тип исключения.

**Квалификации:**

Нет. Примечания:

Нет.

---

**Описание:**

БРОСАТЬ

THROW (Throw Soft Exception) приводит к тому, что текущий контекст устанавливается в контекст обработчика исключений, а ПК переходит к серверному коду обработчика исключений. Кроме того, текущий идентификатор контекста сохраняется в регистре идентификаторов исключенных контекстов. Пользователь может накладывать дополнительные соглашения поверх базовой семантики THROW; например, пользователь может потребовать, чтобы q127 содержал идентификатор мягкого исключения.

**Операция:**

ПК      ПК + 1

Идентификатор исключенного контекста    идентификатор  
контекста    идентификатор контекста      идентификатор  
обработчика исключений

ПК запуск кода сервера обработчика исключений

**Исключения:**

Никто.

**Квалификации:**

Нет. Примечания:

Обратите внимание, что сохраненный ПК не требуется, поскольку ПК исключенного контекста не перезаписывается ПК обработчика исключений: идентификатор контекста устанавливается на обработчик исключений до изменения ПК.

Это многоцикловая инструкция с переменной длительностью выполнения.

ЭКСТАГ контроль качества, контроль качества

EXTAG (Extract Tag) извлекает биты тега из qa и помещает их в qc. Биты тега помещаются в MSB qc и дополняются нулями справа. Область тега фрагмента данных включает верхние 16 бит, тогда как область тега для возможности включает верхние 45 бит. Тип результата в qc – word.

---

**Описание:**

Операция EXTAG выполняется только в том случае, если qa доступен и нет противодействия на qc. В противном случае инструкция останавливается.

**Операция:**

```
если (тип (qa) == возможность) qc
    qa[79:55], 39'b0
иначе qc qa[79:64], 48'b0 тип(qc) слово
```

**Исключения:**

Никто.

**Квалификации:**

Нет. Примечания:

Нет.

SETTAG                      кА, кБ, кК

SETTAG (Установить тег) устанавливает тег данных в qb на значение LSB qa и помещает результат в qc. Это очень мощный оператор, так как он может принудительно выполнить буквальную бинарную трансмутацию типов данных и изменить несколько важных атрибутов фрагмента данных. Если значение битов в qa соответствует возможности, тип qb также должен быть возможностью, и бит владельца для qb должен быть установлен. qa должен иметь тип word.

Операция SETTAG выполняется только в том случае, если доступны оба операнда qa и qb и нет противодействия на qc. В противном случае инструкция останавливается.

**Операция:**

```
если (тип (qa) == слово) если (тип (qb)
    == возможность)
    если (!владелец (qb))
        исключение операции throw
    другие теги(qb) qa[63:39] еще
    если(qa[63] == 1)
        исключение операции throw
```

---

**Описание:**

```
else tags(qb) qa[63:48] else  
выдать исключение типа
```

**Исключения:**

Исключение операции, тип исключения.

**Квалификации:**

Нет. Примечания:

Нет.

РАСПРЕДЕЛИТЬ           кА, кБ, кК

ALLOCATE (выделение возможности) создает возможность qc с размером, наиболее близким к количеству слов, указанных в qb. Адрес возможности и бит «только приращение» устанавливаются для ограничения доступной части возможности точно до размера, указанного в qb. qb должен иметь тип word. Если выделение не удастся, qc возвращается как недопустимая возможность, и выдается исключение нехватки памяти. qa содержит метрику выделения, которая указывает, где в системе следует разместить выделенную память. qa должен иметь тип упакованного char или возможности. Если qa является возможностью, система пытается выделить новую возможность, близкую к возможности в qa.

Операция ALLOCATE выполняется только в том случае, если qa доступен и нет противодействия на qc.

В противном случае выполнение инструкции зайдет в тупик.

**Операция:**

```
если (тип(qb) == слово && (тип(qa) == упакованный символ || (тип(qa) ==  
возможность)))
```

```
если (какие слова доступны)
```

```
контроль качествавозможность размера qa байт
```

```
еще
```

```
контроль качества
```

```
недопустимая возможность
```



---

выбросить исключение из  
памяти  
в противном случае  
выдать исключение  
типа

**Исключения:**

Исключение нехватки памяти, тип исключения.

**Квалификации:**

Нет.

**Примечания:**

Эта инструкция может занять переменное количество циклов для завершения. Эта инструкция является «ленивой» инструкцией.

Формат метрики распределения зависит от реализации. Текущая схема реализации требует, чтобы упакованный символ содержал следующие шестнадцатититные значения символа, от MSB до LSB: игнорируется, игнорируется, ожидаемая частота связи, желаемая задержка.

АЛЛОКАТЕК            qa, n, qc

ALLOCATEC (Allocate Capability, Size in Constant Field) создает возможность qc с размером, наиболее близким к количеству слов, указанному в n. Адрес возможности и бит «только приращение» устанавливаются для ограничения доступной части возможности точно до размера, указанного в n. Если выделение не удастся, qc возвращается как недопустимая возможность, и выдается исключение нехватки памяти. qa содержит метрику выделения, которая указывает, где в системе следует разместить выделенную память. qa должен иметь тип упакованного символа или тип возможности. Если qa является возможностью, система пытается выделить новую возможность, близкую к возможности в qa.

Операция ALLOCATEC выполняется только в том случае, если нет противодействия на qc. В противном случае инструкция останавливается.

**Операция:**

если (тип (qa) == упакованный символ || тип (qa) == возможность)

---

**Описание:**

если (доступно n слов)  
контроль качества  
возможность размера n байт  
еще  
контроль качества      недопустимая  
возможность выбросить исключение из  
памяти  
в противном случае выдать  
исключение типа

**Исключения:**

Исключение нехватки памяти, тип исключения.

**Квалификации:**

Нет. Примечания:

Эта инструкция может занять переменное количество циклов для завершения. Эта инструкция является «ленивой» инструкцией.

Формат метрики распределения зависит от реализации. Текущая схема реализации требует, чтобы упакованный символ содержал следующие шестнадцатититные значения символа, от MSB до LSB: игнорируется, игнорируется, ожидаемая частота связи, желаемая задержка.

ММЛ                      qa, qb

ММЛ (Map Memory Load) сопоставляет номер очереди, указанный в qa, с очередью адресов загрузки и сопоставляет возвращаемые данные загрузки с номером очереди, указанным в qb. qa и qb должны иметь тип word.

Подсистема памяти ожидает, что первый адрес, введенный в очередь адресов памяти, будет возможностью доступа, а последующие записи в очередь адресов загрузки будут смещениями относительно начальной возможности. Постановка в очередь возможности инициализации не приводит к тому, что подсистема памяти возвращает значение загрузки. Если возможность отправляется в подсистему памяти после возможности инициализации, новая возможность поглощает старую; снова, никакое значение загрузки не возвращается в ответ на отправку этой возможности загрузки.

Эта операция останавливается до тех пор, пока qa и qb не будут содержать значение.

**Операция:**

---

```
если (тип(qa, qb) == слово)
    MAP (qa & 0x7F) в очередь адресов загрузки памяти
    Загрузка памяти MAP возвращает очередь данных в (qb &
    0x7F), в противном случае выдает исключение типа
```

**Исключения:**

Тип исключения.

**Квалификации:**

Нет. Примечания:

Эта инструкция может занять переменное количество циклов для завершения отображения, но ПК разрешено увеличивать за один цикл. Это не приводит к неправильной работе, если только пользователь не отменит отображение инструкции по отображению памяти, а затем немедленно переотобразит отображение памяти. Пользователи должны избегать отмены отображения и переотображения карт памяти с использованием тех же очередей в том же контексте. Обратите внимание, что совершенно безопасно повторно инициализировать существующее отображение памяти, отправив новую возможность в очередь адресов.

При отмене сопоставления пары очередей, отображенных в памяти, пользователь несет ответственность за отмену сопоставления как адреса, так и очереди данных. Нет ничего принципиально неправильного в отмене сопоставления только одной очереди; однако это может привести к путанице, если сопоставление очереди будет использоваться повторно, и сборщик мусора не будет отменять распределение памяти, которая имеет хотя бы частичное сопоставление с его возможностями.

MMC                      qa, qb

MMS (Map Memory Store) сопоставляет номер очереди, указанный в qa, с очередью адресов хранения и сопоставляет номер очереди, указанный в qb, с очередью данных хранения. qa и qb должны иметь тип word.

Данные и адреса могут быть поставлены в очередь в разное время и с разной скоростью, но неизменно то, что хранилище блокируется до тех пор, пока в обеих очередях не будет хотя бы одного элемента, и что пары данных и адресов строго коррелируют по их относительному порядку в очередях.

Подсистема памяти ожидает, что первый адрес, введенный в очередь адресов памяти, будет возможностью доступа; этот первый доступ не совпадает с элементом данных в очереди данных хранилища. Последующие

---

**Описание:**

адреса затем интерпретируются как смещения к первоначальной возможности доступа и сопоставляются со значениями данных в очереди данных хранилища.

Эта операция останавливается до тех пор, пока *qa* и *qb* не будут содержать значение.

**Операция:**

```
если (тип (qa, qb) == слово)
    МАР (qa & 0x7F) в очередь адресов памяти
    МАР (qb & 0x7F) в очередь данных для хранения в
    памяти, в противном случае выдать исключение типа
```

**Исключения:**

Тип исключения.

**Квалификации:**

Нет. Примечания:

Эта инструкция может занять переменное количество циклов для завершения отображения, но ПК разрешено увеличивать за один цикл. Это не приводит к неправильной работе, если только пользователь не отменит отображение инструкции по отображению памяти, а затем немедленно переотобразит отображение памяти. Пользователи должны избегать отмены отображения и переотображения карт памяти с использованием тех же очередей в том же контексте. Обратите внимание, что совершенно безопасно повторно инициализировать существующее отображение памяти, отправив новую возможность в очередь адресов.

При отмене сопоставления пары очередей, отображенных в памяти, пользователь несет ответственность за отмену сопоставления как адреса, так и очереди данных. Нет ничего принципиально неправильного в отмене сопоставления только одной очереди; однако это может привести к путанице, если сопоставление очереди будет использоваться повторно, и сборщик мусора не будет отменять распределение памяти, которая имеет хотя бы частичное сопоставление с его возможностями.

Описание:

ОБМЕН

EXCH (Declare Exchange Tuple) отмечает номера очередей, указанные в qa, qb и qc, как кортеж обмена памятью. qa устанавливается как очередь адресов, qb устанавливается как очередь данных в очереди, а qc устанавливается как очередь данных из очереди. Все qa, qb и qc интерпретируются как непосредственные константы. Кортеж обмена должен быть инициализирован путем перемещения возможности в qa до перемещения смещения адреса в qa.

После инициализации кортежа значением адреса следующий фрагмент данных, перемещенный в qb, атомарно обменивается с содержимым памяти по указанному адресу, а содержимое области памяти до обмена помещается в qc.

Система памяти гарантирует, что эта операция будет атомарной на стороне памяти; однако никакие другие относительные тайминги не гарантируются.

Отображение EXCH остается в силе до тех пор, пока оно не будет отменено инструкцией UNMAPQ. Пользователь должен отменить отображение всех трех отображений.

Операция:

MAP qa в очередь адресов атомарной памяти  
MAP qb в очередь входящих данных атомарной памяти MAP qc  
в очередь возвращаемых данных атомарной памяти

Исключения:

Исключение типа и исключение обмена.

Квалификации:

Нет. Примечания:

Выполнение этой инструкции может занять разное количество циклов.

**Описание:****СПАУН**

SPAWN (Spawn) запускает новый поток, выделяя пространство для потока, создавая запись в планировщике потоков для потока с PC, установленным на значение в qb, и возвращая идентификатор потока (который также является возможностью для данных потока) в qc. Разрешения возможности идентификатора потока устанавливаются как непрозрачные и не как владелец. qa содержит метрику порождения, которая используется для руководства временем выполнения относительно того, где должен быть порожден поток. Если qa является возможностью, система пытается выделить новый поток близко к возможности в qa. qa должен иметь тип упакованного символа или тип слова, а qb должен иметь тип слова.

Операция SPAWN выполняется только если qa доступен и нет противодействия на qc. В противном случае инструкция останавливается.

**Операция:**

```
если(тип(qb) == слово && (тип(qa) == упакованный символ || (тип(qa) == возможность)))
```

```
    контроль качества новая  
    возможность потока if(qc ==  
    invalid)
```

```
        выбросить исключение нехватки памяти
```

**Описание:**

в противном случае создайте запись планировщика потоков  
(новый идентификатор потока, ПК = qа)  
в противном случае  
выдать исключение  
типа

**Исключения:**

Тип исключения: Исключение нехватки памяти.

**Квалификации:**

Нет.

**Примечания:**

Эта инструкция может занять различное количество циклов для завершения. Это «ленивая» инструкция.

Формат метрики порождения зависит от реализации. Текущая схема реализации требует, чтобы упакованный символ содержал следующие шестнадцатибитные значения символа, от MSB до LSB: ожидаемые потомки, требования к памяти, требования к вычислениям, желаемая задержка.

**СПАУНЛ**

SPAWNЛ (Load Code and Spawn) запускает новый поток, выделяя пространство для потока, загружая его код, указанный в qЬ, в пространство кода и создавая запись в планировщике потоков для потока с РС, установленным на значение в qа, и возвращая идентификатор потока (который также является возможностью для данных потока) в qс. Разрешения возможности идентификатора потока устанавливаются как непрозрачные, а не как владелец. Размер пространства, выделяемого для потока, кодируется в коде операции OSIZE, который должен быть первой инструкцией нового потока.

**Описание:**

qa должен иметь тип word, а qb должен быть возможностью для массива символов, описывающего универсальный локатор для ресурса кода.

Операция SPAWNL выполняется только в том случае, если доступны оба операнда qa и qb и нет противодействия на qc. В противном случае инструкция останавливается.

**Операция:**

```
если (тип (qa) == слово && тип (qb) == возможность)
    загрузить код, указанный qb, в кодовое пространство qc    возможность
размера, указанного в коде операции OSIZE по адресу в qa if(qc == invalid)
    выбросить исключение нехватки памяти
в противном случае создайте запись планировщика потоков (новый
идентификатор потока, ПК = qa)
в противном случае выдать
исключение типа
```

**Исключения:**

Тип исключения: Исключение нехватки памяти.

**Квалификации:**

Нет. Примечания:

Выполнение этой инструкции может занять разное количество циклов.



---

**Описание:**

СПАУНК

qa, n, qc

SPAWNC (Spawn with PC-constant offset) запускает новый поток, выделяя пространство для потока, создавая запись в планировщике потоков для потока с PC, установленным на значение  $PC + 1 + n$ , и возвращая идентификатор потока (который также является возможностью для данных потока) в qc. Разрешения возможности идентификатора потока устанавливаются как непрозрачные, а не как владелец. Размер пространства, выделяемого для потока, кодируется в коде операции OSIZE, который должен быть первой инструкцией нового потока. qa содержит метрику порождения, которая используется для руководства временем выполнения относительно того, где должен быть порожден поток. Если qa является возможностью, система пытается выделить новую возможность, близкую к возможности в qa.

Операция SPAWNC выполняется только в том случае, если нет противодействия на qc. В противном случае инструкция останавливается.

**Операция:**

если (тип (qa) == упакованный символ || тип (qa) == возможность)

    контроль качества возможность размера в коде операции OSIZE

    в  $(n + PC + 1)$  if(qc == invalid)

        выбросить исключение нехватки памяти

    в противном случае создать запись планировщика потоков (новый идентификатор потока,  $PC = n + PC + 1$ ) в противном случае выдать исключение типа

**Исключени****я:**

Исключение нехватки памяти и исключение типа.

**Квалифика****ции:**

---

**Описание:**

Нет.

**Примечания**

:

Эта инструкция может занять различное количество циклов для завершения. Это «ленивая» инструкция.

Формат метрики порождения зависит от реализации. Текущая схема реализации требует, чтобы упакованный символ содержал следующие шестнадцатититные значения символа, от MSB до LSB: ожидаемые потомки, требования к памяти, требования к вычислениям, желаемая задержка.

MAPQ                      кА, кБ, кК

MAPQ (Map Queue) — это инструкция специального формата. qa фактически интерпретируется как непосредственная константа: она указывает номер очереди в текущем контексте, который должен быть отображен. MAPQ фактически не считывает и не изменяет содержимое qa каким-либо образом. Модификатор копирования/затирания не влияет на значение qa в этом случае. qb указывает номер очереди для чтения для номера очереди отображения назначения, а qc указывает номер очереди для чтения для идентификатора контекста назначения.

Операция MAPQ выполняется только в том случае, если доступны оба операнда qb и qc.

**Операция:**

если (тип (qb) == слово && тип (qc) ==  
возможность) сопоставить очередь 'qa'.tail в  
текущем контексте с очередью ((qb & 0x7F)  
7).head в контексте qc, иначе выдать  
исключение типа

**Исключени**

я:

---

**Описание:**

Тип исключения.

**Квалифика  
ции:**

Нет.

**Примечания**

:

Необычный формат этой инструкции является артефактом обратной совместимости с более ранней версией набора инструкций. Эта инструкция может быть представлена внутри аппаратной реализации более типичным образом и потребовать от ассемблера выполнения простого преобразования формата. Для завершения этой инструкции может потребоваться несколько циклов.

MAPQC

кА, кБ, кК

MAPQC (Map Queue with Destination as Constant) — это инструкция специального формата. qa и qb фактически интерпретируются как непосредственные константы: они указывают номер очереди в текущем контексте и номер очереди назначения, соответственно, которые должны быть отображены. MAPQC фактически не считывает и не изменяет содержимое qa или qb каким-либо образом. Модификатор copy/clobber не влияет на значение qa и qb в этом случае. qc указывает номер очереди для чтения для идентификатора контекста назначения.

Операция MAPQC выполняется только при наличии операнда qc.

**Операция:**

```
если (тип (qc) == возможность)
    сопоставить очередь 'qa'.tail в текущем
    контексте с очередью 'qb'.head в контексте qc
в противном случае
    выдать исключение
    типа
```

---

Описание:

Исключени

я:

Тип исключения.

Квалифика

ции:

Нет.

Примечания

:

Необычный формат этой инструкции является артефактом обратной совместимости с более ранней версией набора инструкций. Эта инструкция может быть представлена внутри аппаратной реализации более типичным образом и потребовать от ассемблера выполнения простого преобразования формата. Для завершения этой инструкции может потребоваться несколько циклов.

МАПСQ

qa, qb

MAPSQ (Map Queue Source) — это инструкция специального формата. qa и qb фактически интерпретируются как непосредственные константы. MAPSQ создает отображение таким образом, что каждый элемент, помещенный сетевым интерфейсом в очередь, указанную в немедленной константе qa, также помещает контекстный идентификатор источника данных в очередь, указанную в немедленной константе qb. Поступление данных из сетевого интерфейса в очередь, указанную qa, гарантированно происходит одновременно с поступлением контекстного идентификатора в очередь, указанную qb. Результирующим типом идентификаторов в qb является возможность, с установленным битом непрозрачности и очищенным битом владельца.

Операция:

сопоставить идентификатор источника входящих данных очереди (qa и 0x7F) с (qb и 0x7F)

---

Описание:

Исключени

я:

Никто.

Квалифика

ции:

Нет.

Примечания

:

Выполнение этой инструкции может занять разное количество циклов.

Обратите внимание, что данные, поступающие в qa через локальные операции, не приводят к тому, что qb ставит источник в очередь; таким образом, не рекомендуется совместно использовать qa как цель для локальных и удаленных операций.

MAPDROP                      qa

MAPDROP (Set Mapping to Drop Mode) — это инструкция специального формата, в которой qa интерпретируется как непосредственная константа. MAPDROP устанавливает режим отображения номера очереди, указанного непосредственная константа qa, в режим «drop». В этом режиме обратное давление на очередь приводит к отбрасыванию данных вместо остановки контекста. Это особенно полезно при реализации операторов чистого потокового вещания для типов данных реального времени, таких как видео или аудио.

Операция:

установить режим очереди (qa и 0x7F) на режим сброса

Исключени

я:

Никто.

Квалифика

ции:

---

Описание:

Нет.

Примечания

:

Выполнение этой инструкции может занять разное количество циклов.

---

**Описание:**

UNMAPQ                      qa

UNMAPQ (Unmap A Queue) — это специальная инструкция формата, в которой qa интерпретируется как непосредственная константа. UNMAPQ сбрасывает отображение очереди, указанной непосредственной константой qa, на значение по умолчанию (текущий идентификатор контекста). Следует позаботиться о том, чтобы гарантировать, что указанная очередь пуста, прежде чем выдавать эту инструкцию, в противном случае оставшиеся данные, которые могут находиться в очереди, когда эта инструкция удаляется, никогда не будут доставлены по назначению.

**Операция:**

установить сопоставление очереди (qa & 0x7F) с текущим идентификатором контекста

**Исключени  
я:**

Никто.

**Квалифика  
ции:**

Нет.

**Примечания**

:

Выполнение этой инструкции может занять разное количество циклов.

При отмене сопоставления пары очередей, отображенных в памяти, пользователь несет ответственность за отмену сопоставления как адреса, так и очереди данных. Нет ничего принципиально неправильного в отмене сопоставления только одной очереди; однако это может привести к путанице, если сопоставление очереди будет использоваться повторно, и сборщик мусора не будет отменять распределение памяти, которая имеет хотя бы частичное сопоставление с его возможностями.

Описание:

ПОСЫЛКА кА, кБ, кК

 $\kappa_A, \kappa_B, \kappa_K$ 

PARCEL (Parcel out a Capability) берет возможность в qa и пытается создать подвозможность с адресом и тегами, описанными в qb. Результат помещается в qc. qa должна быть возможностью, qb – типом слова, а результат qc – возможностью. Формат адреса подвозможности и спецификатора тега – 15 бит тегов, за которыми следует 1-битное поле только для приращения, за которым следует 35-битное поле адреса. Неиспользуемые биты слева игнорируются.

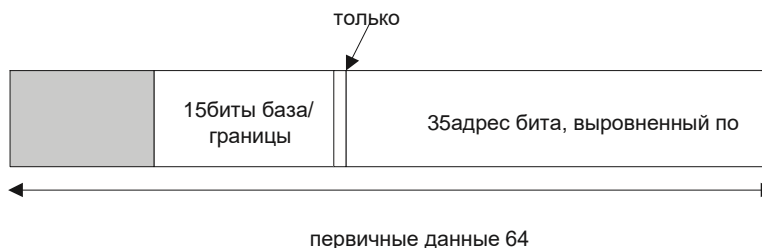


Рисунок D-1: формат qb для инструкции PARCEL

Если возможность, описанная  $qb$ , выходит за рамки заданной возможности в  $qa$ , выдается исключение операции, а результат в  $qc$  становится недопустимым.

Операция **PARCEL** выполняется только в том случае, если доступны оба операнда **qa** и **qb** и нет противодействия на **qs**. В противном случае инструкция останавливается.

Операция:

```
если (тип (qa) == возможность && тип (qb) == слово)
    контроль качества подвозможность qa,
    описанная qb else throw type exception
```

## Исклучени

Я:



---

**Описание:**

Исключение типа и исключение операции.

**Квалифика  
ции:**

Нет.

**Примечания**

:

Выполнение этой инструкции может занять разное количество циклов.

МСИНК

MSYNC (синхронизация памяти) останавливает текущий поток до тех пор, пока не будут завершены все ожидающие выполнения операции с памятью текущего потока.

**Операция:**

если (текущий поток имеет ожидающие операции с памятью)

ПК ПК

сигнал структурного останова для планировщика потоков

еще

ПК ПК + 1

**Исключени**

**я:**

Никто.

**Квалифика**

**ции:**

Нет.

**Примечания**

:

Для выполнения этой инструкции потребуется разное количество циклов.

---

**Описание:**

LDCODE контроль качества, контроль качества

LDCODE (Dynamically Load Code) берет возможность в `qa`, которая содержит массив символов, который называет объект кода и его путь, пытается загрузить его в память кода и возвращает абсолютный адрес ПК кода как слово в `qc`. Неудача при завершении этой операции приводит к возникновению исключения загрузки кода и недопустимости `qc`.

(Необходимо определить, должно ли возвращаться значение ПК или это должен быть идентификатор контекста для запущенного сервера объектов...)

Операция LDCODE выполняется только в том случае, если `qa` доступен и нет противодействия на `qc`.

В противном случае выполнение инструкции зайдет в тупик.

**Операция:**

```
если (тип (qa) == возможность)
    если (qa.permissions == чтение, не непрозрачно, допустимо)
        загрузочный код, описанный массивом
        символов в qa qc ПК точки входа кода
        if(tags(qc) == invalid)
            выдать исключение загрузки кода
        в противном случае выдать
        исключение операции
    в противном случае
        выдать исключение
        типа
```

**Исключени**

**я:**

Исключение типа, исключение операции и исключение загрузки кода.

**Квалифика**

**ции:**

---

Описание:

Нет.

Примечания

:

Выполнение этой инструкции может занять разное количество циклов.

ОСАЙЗ

н

OSIZE (Object Size Directive) — это директива компилятора, которая использует формат кода операции «подсказки» для информирования ADAM о том, какой размер области необходимо выделить для конкретного объекта потока. Размер области для выделения в словах указывается в *n*. Этот код операции может располагаться где угодно, но он имеет значение только тогда, когда находится в последовательности инструкций точки входа для кода инициализатора объекта. При выполнении эта инструкция не делает ничего с состоянием машины, кроме увеличения PC.

Операция:

ПК = ПК + 1

Исключени

я:

Никто.

Квалифика

ции:

Нет.

Примеч

ания:

Нет.

---

**Описание:**

НАМЕКАТЬ                      t,подсказка

HINT (Compiler Hint) — это подсказка от компилятора или программиста для системы выполнения ADAM. Инструкция HINT не влияет на состояние машины ADAM, за исключением увеличения PC; однако она может оказать глубокое влияние на ОС и/или сопроцессор управления.

Тип подсказки кодируется в поле t, а фактическое значение подсказки кодируется в поле подсказки. Допустимые типы подсказок TBD, но они делятся на две широкие категории: машинно-зависимые и машинно-независимые. Машинно-зависимые подсказки включают директивы размещения данных. Машинно-независимые подсказки включают подсказки обмена потоками, директивы предварительной выборки и подсказки миграции. Подсказка с нераспознанным типом подсказки игнорируется.

Неправильная подсказка никогда не приводит к неправильным результатам работы программы; неправильная подсказка приводит лишь к снижению производительности.

**Операция:**

ПК = ПК + 1

**Исключени**

**я:**

Никто.

**Квалифика**

**ции:**

Нет.

**Примеч**

**ания:**

Нет.

НОП

---

**Описание:**

NOP (нет операции) Инструкция NOP не оказывает никакого влияния на состояние машины ADAM, за исключением увеличения PC.

**Операция:**

$ПК = ПК + 1$

**Исключени  
я:**

Никто.

**Квалифика  
ции:**

Нет.

**Примеч**

**ания:**

Нет.

# Библиография

- [АБ93] Арвинд и Стивен Бробст. Эволюция архитектур потоков данных: от статического потока данных до P-RISC. Международный журнал высокоскоростных вычислений, 5(2), 1993.
- Всемирная научная издательская компания.
- [АВС 95] A. Agarwal, R. Bianchini, D. Chaiken, KL Johnson, D. Kranz, J. Kubiatawicz, BH Lim, K. Mackenzie и D. Yeung. Машина MIT alewife: архитектура и производительность. В трудах 22-го Международного симпозиума по компьютерной архитектуре, страницы 2–13, Санта-Маргерита-Лигуре, Италия, июнь 1995 г.
- [АНКВ00] Викас Агарвал, М. С. Хришикеш, Стивен В. Кеклер и Дуг Бергер. Тактовая частота против IPC: конец пути для обычных микроархитектур. В трудах 27-го ежегодного международного симпозиума по компьютерной архитектуре. ACM, 2000.
- [АКК 95] Гейл Алверсон, Саймон Кахан, Ричард Корри, Кэти Макканн и Бертон Смит.
- Планирование на Tera MTA. В книге Дрора Г. Фейтельсона и Ларри Рудольфа, редакторов, «Стратегии планирования заданий для параллельной обработки», номер 949 в LNCS, страницы 19–44, 1995.
- [АЛКК91] A. Agarwal, BH Lim, D. Kranz и J. Kubiatawicz. LimitLESS directorys: A scalable cache coherence scheme. В Proceedings of ASPLOS IV, страницы 224–234, апрель 1991 г.
- [АШН88] A. Agarwal, R. Simoni, J. Hennessy и M. Horowitz. Оценка схем каталогов для обеспечения когерентности кэша. В трудах 15-го Международного симпозиума по
- Архитектура компьютера*, страницы 28–289, июнь 1988 г.
- [БЕЙ98] Аллан Бородин и Ран Эль-Янив. *Онлайн-вычисления и конкурентный анализ*. Издательство Кембриджского университета, 1998.
- [БГКХ00] Джереми Х. Браун, Дж. П. Гроссман, Том Найт и Эндрю Хуан. Возможность-ity представление со встроенным адресом и почти точными границами объекта. Технический отчет Project Aries Tech Note 5, Массачусетский технологический институт AI Lab,

<http://www.ai.mit.edu/projects/aries>, 2000.

[БЛ94] Р. Блюмоф и К. Лейзерсон. Планирование мультпотокковые вычисления по работе кражи-

ing. В трудах 35-го ежегодного симпозиума по основам компьютерной науки, Санта-Фе, Нью-Мексико, страницы 356–368, ноябрь 1994 г.

[БП92] Джефф Бакстер и Джанак Х. Патель. Профилирование на основе миграции задач. *ВШестой Интернационал*

*Симпозиум IEEE по параллельной обработке*, страницы 192–195, 1992.

[Бро02] Джереми Хэнфорд Браун. *Редкограничные массивы: механизм, поддерживающий паралл-*

*Распределение, коммуникация и сбор мусора*. Кандидатская диссертация, Массачусетский технологический институт, 2002.

[КЧ 00] Эйлон Каспи, Майкл Чу, Рэнди Хуанг, Джозеф Йе, Джон Вавжинек и Андре DeHon. Потокковые вычисления, организованные для реконфигурируемого выполнения (SCORE). На конференции по программируемой на поле логике и приложениям. Springer-Verlag, август 2000 г.

[CDV 94] Рохит Чандра, Скотт Девайн, Бен Вергезе, Ануп Гупта и Мендель Розенблюм. Планирование и миграция страниц для многопроцессорных вычислительных серверов. В трудах ASPLOS VI, Сан-Хосе, Калифорния, 1994.

[Че] Эрик Чивер. Методы Рунге-Кутты. <http://www.swarthmore.edu/natsci/echeever1/Ref/NumericInt/RK2.html>.

[CI00a] Международный комитет дорожной карты и ITWG. Международная дорожная карта для полупроводников

Обновление ductors. Технический отчет, SIA, <http://public.itrs.net/>, 2000.

[CI00b] Международный комитет дорожной карты и ITWG. Международная дорожная карта для полупроводников

ductors, обновление PIDS. Технический отчет, SIA, <http://public.itrs.net/>, 2000.

- [CI00c] Международный комитет дорожной карты и ITWG. Международная дорожная карта для полупроводников  
воздуховоды, обновление конструкции. Технический отчет, SIA,  
<http://public.itrs.net/>, 2000.
- [CJDM01] V. Cuppu, B. Jacob, B. Davis и T. Mudge. Высокопроизводительные DRAM в рабочих станциях. IEEE Transactions on Computers, 50(11):1133–1153, ноябрь 2001 г.
- [CKD94] Николас П. Картер, Стивен В. Кеклер и Уильям Дж. Далли. Аппаратная поддержка быстрой адресации на основе возможностей. В трудах ASPLOS VI, октябрь 1994 г.
- [CKW96] Оливер Чупке, Дитмар Коттманн и Ханс-Дирк Вальтер. Миграция объектов в монолитных распределенных приложениях. В трудах 16-й Международной конференции по  
*Распределенные вычислительные системы*, страницы 529–536, 1996.
- [CM97] Хейнс Д. Кронк и П. М. Мехротра. Миграция потоков при наличии указателей. В трудах 30-й Гавайской международной конференции по системным наукам, том 1, страницы 292–298. IEEE, 1997.
- [Кора] МоСис Корпорация. TSMC 0,13 м процесс быстрый 1-T SRAM краткое содержание.  
[http://www.mosys.com/1t\\_sr.html](http://www.mosys.com/1t_sr.html). Для доступа к материалам дизайна требуется регистрация.  
другие.
- [Корб] Тайваньская корпорация по производству полупроводников. 0,18 м. Краткое изложение процесса.  
<http://www.tsmc.com/technology/cl018.html>.
- [CS99] Дэвид Э. Каллер и Джасвиндер Пал Сингх. Архитектура параллельного компьютера: аппаратно-программный подход. Morgan Kaufmann Publishers, 1999.
- [CSS 91] D. Culler, A. Sah, K. Schauzer, T. von Eicken и J. Wawrzynek. Мелкозернистый параллелизм с минимальной аппаратной поддержкой: абстрактная машина с потоками, управляемая компилятором. В трудах 4-й Международной конференции по архитектурной поддержке языков программирования и операционных систем, Санта-Клара, Калифорния, апрель 1991 г.
- [DeH90] Андре ДеХон. Маршрутизация по древовидной структуре для транзита. Технический отчет по ИИ 1224 диссертации SB, Лаборатория искусственного интеллекта Массачусетского технологического института, 1990.



- [DeH93] Андре ДеХон. Надежная, высокоскоростная сетевая конструкция для крупномасштабной многопроцессорной обработки. Технический отчет по ИИ 1445 диссертации SM, Лаборатория искусственного интеллекта Массачусетского технологического института, 200 Technology Square, Кембридж, Массачусетс 02139, 1993.
- [DS90] М. Дюбуа и К. Шойрих. Зависимости доступа к памяти в мультипроцессорах с общей памятью. Труды IEEE по программной инженерии, 16(6):660–673, 1990.
- [ea92] Х. Буркхарт III и др. Обзор компьютерной системы KSR1. Технический отчет KSR-TR-9202001, Kendall Square Research, Бостон, февраль 1992 г.
- [ФКД 95] Марко Филло, Стивен В. Кеклер, Уильям Дж. Далли, Николас П. Картер, Эндрю Чанг, Евгений Гуревич и Уэй С. Ли. Мультикомпьютер M-Machine. В трудах 28-го ежегодного международного симпозиума по микроархитектуре, страницы 146–156. ИИЭР, 1995.
- [ФНН93] Мэтью К. Фарренс, Пиус Нг и Фил Нико. Сравнение суперскалярной и разьединенной архитектур доступа/исполнения. В трудах 26-го ежегодного международного симпозиума по микроархитектуре, страницы 100–103. IEEE, 1993.
- [Гал96] Майк Галлес. Микросхема-паук SGI. В Proceedings of Hot Interconnects IV, страницы 141–146. IEEE, 1996.
- [GB02] JP Grossman и Jeremy Brown. Легкий идемпотентный протокол обмена сообщениями для неисправных сетей. В выступлении в Трудах Симпозиума по Параллельным Алгоритмам и архитектурам, 2002.
- [GBHK00] JP Grossman, Jeremy Brown, Andrew Huang и Tom Knight. Использование SQUID для решения проблемы псевдонимов указателей пересылки. Технический отчет Project Aries Tech Note 4, Массачусетский технологический институт AI Lab, <http://www.ai.mit.edu/projects/aries>, 2000.
- [ГХИ94] М. Гокхале, Б. Холмс и К. Иобст. Обработка в памяти: массив массивов параллельной PIM Terasys. IEEE Computer, 28(4):23–31, 1994.

- [Гро01] Дж. П. Гроссман. Проектирование и оценка параллельного компьютера Namal. Предложение по докторской диссертации, Массачусетский технологический институт, 2001.
- [GWM90] А. Гупта, В. Д. Вебер и Т. Моури. Сокращение требований к памяти и трафику для масштабируемые схемы когерентности кэша на основе каталогов. В Трудах Международной конференции по параллельной обработке, страницы 312–321, август 1990 г.
- [Хар00] Шон Р. Хартсок. Гравитация работает! [http://modzer0.cs.uaf.edu/hartsock/C\\_Spp/OpenGL/Gravity.html](http://modzer0.cs.uaf.edu/hartsock/C_Spp/OpenGL/Gravity.html), март 2000 г. Первоначально найдено на [www.planetsource-code.com](http://www.planetsource-code.com).
- [ННК 01] Джозеф Холл, Джейсон Хартлайн, Анна Р. Карлин, Джаред Сайя и Джон Уилкс. Об алгоритмах эффективной миграции данных. В Двенадцатом ежегодном симпозиуме по дискретным алгоритмам, страницы 620–629. ACM SIGACT и SIAM, 2001.
- [XXC 00] Лэнс Хаммонд, Бен Хабберт, Майкл Сиу, Манохар Прабху, Майк Чен и Кунле Олукотун. SMP гидры Стэнфорда. Журнал IEEE Micro Magazine, март–апрель 2000 г.
- [HS94a] Чао-Джу Хоу и Кан Г. Шин. Проектирование и оценка эффективного распределения нагрузки в распределенных системах реального времени. Труды IEEE по параллельным распределенным системам, 5(7):704–719, июль 1994 г.
- [HS94b] Чао-Джу Хоу и Кан Г. Шин. Распределение нагрузки с учетом будущих поступлений задач в гетерогенных распределенных системах реального времени. IEEE Transactions on Computers, 43(9):1076–1090, сентябрь 1994 г.
- [Hsi95] Уилсон Ченг-И Хси. Динамическая миграция вычислений в распределенных системах с общей памятью. Кандидатская диссертация, Массачусетский технологический институт, 1995.
- [ХСУ 01] Гленн Хинтон, Дэйв Сейджер, Майк Аптон, Даррелл Богтс, Дуг Кармин, Алан Кайкер и Патрис Руссель. Микроархитектура процессора Pentium 4. Intel Technology Journal, Q1 2001.  
<http://developer.intel.com/technology/itj/q12001/articles/art.2.htm>.

- [HWW93] Wilson C. Hsieh, Paul Wang и William E. Wehl. Миграция вычислений: Повышение локальности для параллельных систем с распределенной памятью. В трудах Четвертого ACM PPOPP, страницы 239–248, Калифорния, май 1993 г.
- [Иан88] Роберт Ианнуччи. На пути к гибридной архитектуре потока данных/фон Неймана. В трудах 15-го ежегодного международного симпозиума по компьютерной архитектуре, Гонолулу, Гавайи, май 1988 года. Object Management Group Inc., редактор. Общий брокер запросов объектов: [Inc01] Архитектура и спецификация (Версия 2.5). <http://www.omg.org>, сентябрь 2001 г.
- [JLNB88] Эрик Джул, Генри Леви, Норман Хатчинсон и Эндрю Блэк. Мелкозернистые мобильные вычисления в системе Emerald. ACM Transactions on Computer Systems, 6(1):109–133, февраль 1988 г.
- [Джо96] CF Joerg. Система cilk для параллельных многопоточных вычислений. Технический отчет MIT/LCS/TR-701, Лаборатория компьютерных наук Массачусетского технологического института, 1996.
- [KMSM01] В. Калогераки, П. М. Меллиар-Смит и Л. Э. Мозер. Динамические алгоритмы миграции для распределенных объектных систем. В 21 IEEE Международная конференция по распределенным вычислительным системам, страницы 119–126, Феникс, Аризона, апрель 2001 г.
- [КПП 97] К. Козыракис, С. Периссакис, Д. Паттерсон, Т. Андресон, К. Асанович, Н. Кардвелл, Р. Фромм, Дж. Голбус, Б. Грибстад, К. Китон, Р. Томас, Н. Треухафт и К. Йелик. Масштабируемые процессоры в эпоху миллиардов транзисторов: IRAM. IEEE Computer, страницы 75–78, сентябрь 1997 г.
- [LBCF 00] ДжМ Лондон, Э. Баркли, Младший CG Фонтан, А. Лумис из Массачусетского технологического института Линколн Laboratory и Фари Ассадераги из IBM. Кремний на арсениде галлия для оптоэлектронной интеграции. Технический отчет, MIT MTL, <http://www.mtl.mit.edu/mtlhome/6Res/AR2000/AR00index.html>, 2000.

[ЛБФ 98]

Уолтер Ли, Раджив Баруа, Мэтью Фрэнк, Девабхактуни Шрикришна, Джонатан Бабб, Вивек Саркар и Саман Амарасингхе. Пространственно-временное планирование параллелизма на уровне инструкций на машине RAW. В трудах ASPLOS-VIII, Калифорния. ACM, 1998.

[ЛФА96] Дэвид К. Ловенталь, Винсент В. Фри и Грегори Р. Эндрюс. Использование мелкозернистых потоков и принятие решений во время выполнения в параллельных вычислениях. Журнал параллельных и *Распределенные вычисления*, 37(1):41–54, август 1996 г.

[ЛХ94]

Бен Ли и А. Р. Хьюрон. Архитектуры потоков данных и многопоточность. IEEE Computer, август 1994 г.

[ЛЖ90]

Карл Э. Лав и Гарри Ф. Джордан. Исследование статического и динамического расписания *ing*. В трудах 17-го *Ежегодный международный симпозиум по компьютерной архитектуре*, страницы 192–201. IEEE, 1990.

[ЛЛ197] Джеймс Лаудон и Дэниел Леноски. Обзор системы линейки продуктов SGI Origin 200/2000. В COMPCON. IEEE, 1997.

[ЛЛГ 92] Д. Леноски, Дж. Лаудон, К. Гарачорлоо, В. Д. Вебер, А. Гупта и Дж. Хеннесси. Стэнфордский многопроцессорный прибор. IEEE Computer, 25(3):63–79, март 1992 г.

[ЛВ95]

Дэниел Э. Леноски и Вольф-Дитрих Вебер. Масштабируемая многопроцессорная обработка с общей памятью. Морган Кауфманн, 1995.

[Mac00]

International Business Machine. Blue Logic Cu-11 ASIC. Технический отчет SA14–2451–00, IBM, 2000.

[Мар00]

Норман Марголус. Архитектура встроенной DRAM для крупномасштабных пространственно-решеточных вычислений. В 27-м ежегодном международном симпозиуме по компьютерной архитектуре, страницы 149–160. IEEE, 2000.

[МакФ97]

Грант В. Макфарланд. Масштабирование технологии КМОП и его влияние на задержку кэша. Кандидатская диссертация, Стэнфордский университет, июнь 1997 г.

- [MEK 99] CE Molnar, IW Jones, WS Coates, JK Lexau, SM Fairbanks и IE Sutherland. Два эксперимента по производительности кольца FIFO. В Proceedings of the IEEE, том 87, страницы 297–307, февраль 1999 г.
- [MJCL97] CE Molnar, IW Jones, WS Coates и JK Lexau. Эксперимент по проверке производительности кольца FIFO. В трудах Третьего международного симпозиума по перспективным исследованиям в области асинхронных цепей и систем, страницы 279–289, 1997.
- [МПЖ 00] К. Mai, Т. Paaske, N. Jayasena, R. Ho, W. Dally и M. Horowitz. Умные воспоминания: модульная реконфигурируемая архитектура. В трудах 13-го ежегодного международного симпозиума по компьютерной архитектуре, июнь 2000 г.
- [MSAD90] Уильям Манджионе-Смит, Сантош Г. Абрахам и Эдвард С. Дэвидсон. Влияние латентности памяти и мелкозернистого параллелизма на производительность астронавтики ZS-1. В трудах двадцать третьей ежегодной Гавайской международной конференции по системам  
*Науки*, том 1, страницы 288–296. IEEE, 1990.  
Уильям Манджионе-Смит, Сантош Г. Абрахам и Эдвард С. Дэвидсон.
- [MSAD91] Архитектор-  
естественные и фактические характеристики IBM RS/6000 и Astronautics ZS-1.  
*Труды двадцать четвертой ежегодной Гавайской международной конференции по системным наукам*, том 1, страницы 397–408. IEEE, 1991.
- [MSW93] Хенк Л. Мюллер, Пол У. А. Сталлард и Дэвид Х. Д. Уоррен. Машина диффузии данных с масштабируемой сетью точка-точка. Технический отчет CSTR-93-17, факультет компьютерных наук Бристольского университета, октябрь 1993 г.
- [HA89] Ришиюр С. Нихил и Арвинд. Может ли поток данных включить в себя вычисления фон Неймана?  
*Труды 16 Ежегодный международный симпозиум по компьютерной архитектуре*,  
Иерусалим, Израиль, май 1989 г.
- [ND91] Питер Р. Нут и Уильям Дж. Далли. Механизм эффективного переключения контекста. В сборнике International Conference on Computer Design, страницы 301–304, 1991.
- [ND95] Питер Р. Нут и Уильям Дж. Далли. Файл именованного государственного регистра: Реализация и производительность. В Трудях Первого симпозиума IEEE по высокопроизводительным  
*Архитектура компьютера*, страницы 4–13, 1995.

- [НВД93] MD Noakes, DA Wallach и WJ Dally. Мультикомпьютер J-Machine: Архитектурная оценка. В трудах 20-го ежегодного симпозиума по компьютерам *Архитектура*, страницы 224–235, 1993.
- [OCS98] M. Oksin, FT Chong и T. Sherwood. Активные страницы: вычислительная модель для интеллектуальной памяти. В 25-м ежегодном симпозиуме по компьютерной архитектуре, страницы 192–203. IEEE, 1998.
- [ОНХ 96] Кунле Олукотун, Басем А. Найфех, Лэнс Хаммонд, Кен Уилсон и Куньюнг Чанг. Дело в пользу однокристалльного мультипроцессора. В трудах ASPLOS-VII, *Кембридж, Массачусетс*. АСМ, 1996.
- [ПББ93] GA Papadopoulos, Greiner R. Boughton и MJ Beckerle. \*Т: Интегрированные строительные блоки для параллельных вычислений. В трудах конференции по суперкомпьютерам, страницы 623–635, 1993.
- [PM83] Майкл Л. Пауэлл и Бартон П. Миллер. Миграция процессов в DEMOS/MP. В Pro-  
уступки 9 *Симпозиум АСМ по принципам операционных систем*, страницы 110–119, Нью-Йорк, 1983. АСМ Press.
- [RC96] Эллард Т. Рауш и Рой Х. Кэмпбелл. Быстрая динамическая миграция процессов. В Трудах 16-й Международной конференции по распределенным вычислительным системам, страницы 637–645. IEEE, 1996.
- [RSAU91] Ларри Рудольф, Мириам Сливкин-Аллуф и Эли Упфал. Простая схема балансировки нагрузки для распределения задач на параллельных машинах. В ACM Symposium on Parallel Algorithms and Architectures, страницы 237–245, 1991.
- [Щ] Уэйн Шлитт. Xstar n-body решатель. <http://www.midwestcs.com/xstar/>.
- [Sco96] Стивен Л. Скотт. Синхронизация и связь в мультипроцессоре ТЗЕ. В трудах ASPLOS VII, Массачусетс, 1996. АСМ.

[СДВ 87]

JE Smith, GE Dermer, BD Vanderwarn, SD Klinger, CM Rozewski, DL Folwler, KR Scidmore и JP Laudon. Центральный процессор ZS-1. На Второй международной конференции по архитектурной поддержке языков программирования и операционных систем, страницы 199–204, октябрь 1987 г.

[Сэм]

Matrix Semiconductor. Сайт Matrix Semiconductor. <http://www.matrixsemi.com>.

[ШК95]

Бехруз А. Ширази, Али Р. Хурсон и Кришна М. Кави, редакторы. Планирование и

*Балансировка нагрузки в параллельных и распределенных системах.* Издательство компьютерного общества IEEE, 1995.

[СКА01]

Майкл Санг, Ронни Крашински и Крсте Асанович. Многопоточные развязанные архитектуры для сложных вычислений общего назначения. В семинаре по развязанным архитектурам доступа к памяти, РАСТ-01, Барселона, Испания, сентябрь 2001 г.

[Сми82a]

Бертон Смит. Архитектура и применение многопроцессорной вычислительной системы НЕР. В Трудах Международного общества оптической инженерии, страницы

241–248, 1982.

[Smi82b]

Джеймс Э. Смит. Архитектуры компьютеров с отдельным доступом/исполнением. В трудах 9-го ежегодного международного симпозиума по архитектуре компьютеров, Остин, Техас, апрель 1982 г.

[Сте85]

Дэвид Стивенсон. Стандарт IEEE для двоичной арифметики с плавающей точкой. ANSI/IEEE

стандарт 754-1985, август 1985 г.

[ТР96] Хосеп Торреллас и Дэвид Падуа. Проект мультипроцессора агрессивной комы в Иллинойсе

(I-ACOMA). В 6 *Симпозиум по рубежам массовых параллельных вычислений*, Октябрь 1996 г.

[Tuc84]

Д. Б. Такерман. Микроструктуры теплопередачи для интегральных схем. Кандидатская диссертация, Стэнфордский университет, 1984.

[Ван02]

Бенджамин Мид Вандивер. Компиляция в архитектуру на основе очередей. Магистерская диссертация, Массачусетский технологический институт, 2002.

- [vCGS92] Т. фон Эйкен, Д. Каллер, С. Голдштейн и К. Шаузер. Активные сообщения: механизм для интегрированной коммуникации и вычислений, 1992.
- [BG98] Александр В. Вейденбаум и К. А. Галливан. Архитектура DRAM с отдельным доступом. IEEE Innovative Architecture for Future Generation High-Performance Processors and Systems, страницы 94–103, 1997, 1998.
- [WC01] Роберт Вудс-Корвин. Высокоскоростная отказоустойчивая соединительная фабрика для крупномасштабной многопроцессорной обработки. Магистерская диссертация, Массачусетский технологический институт, 2001.
- [WCD 95] Чи-По Вен, Сумен Чакрабартти, Этьен Депри, Арвинд Кришнамурти и Кэтрин Йелик. Поддержка времени выполнения для переносимых распределенных структур данных. В Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers, май 1995 г.
- [WGQH98] Б. Вайсман, Б. Гомес, Дж. Куиттек и М. Хольткамп. Эффективная мелкозернистая резьба  
миграция с активными потоками. В представлении на 12-й Международный симпозиум по параллельной обработке и 9-й Симпозиум по параллельной и распределенной обработке, страницы 410–414, 1998.
- [Вул92] Уильям А. Вульф. Оценка архитектуры WM. В трудах 19-го ежегодного международного симпозиума по архитектуре компьютеров, стр. 382–390. ACM, 1992.
- [XL97] Чэнчжун Сюй и Фрэнсис СМ Лау. Балансировка нагрузки в параллельных компьютерах:  
*теория и практика*. Kluwer Academic Publishers, 1997.