Московский Физико-Технический Институт

Микроархитектура современных микропроцессоров

Homework 3

Автор:

Овсянников Михаил

Долгопрудный, 2025

Содержание

Introduction

Question 1

Приведите формулы для Performance, Power / Dynamic Power, назовите параметры в них входящие

Answer 1

Формула для Performance:

$$Performace = \frac{1}{Time} = \frac{1}{N_{\text{instr}} \cdot CPI \cdot T_{\text{cycle}}} = \frac{1}{N_{\text{instr}}} \cdot IPC \cdot f$$

Здесь:

- N_{instr} – количество инструкций
- CPI – cycle per instruction, то есть количество тактов на одну инструкцию
- T_{cycle} – время одного такта процессора
- IPC – instruction per cycle, то есть количество инструкций за один такт
- f – частота процессора

Формула для Power:

$$P = C \cdot V^2 \cdot f + P_{\text{leakage}}$$

Первая часть являет собой Dynamic Power:

$$P_{\text{dyn}} = C \cdot V^2 \cdot f$$

В последних двух формулах:

- C – общая ёмкость элементов
- V – напряжение
- f – частота
- P_{leakage} – мощность утечек элементов

Question 2

Объясните законы Мура (Moore's law) и Деннарда (Dennard scaling)

Answer 2

Закон Мура: количество транзисторов, размещаемых на кристалле интегральной схемы, удваивается каждые 2 года.

Закон Деннарда: с каждым поколением технологий размеры транзисторов становятся меньше, но потребляемая мощность на единицу площади на кристалле остаётся одинаковой. Напряжение и ток уменьшаются при уменьшении размеров транзисторов.

Question 3

Когда закончился Dennard scaling? Чем знаменуется его окончание?

Answer 3

Dennard Scaling перестал соблюдаться около 2006 года, поскольку мощность утечек не масштабируется с размером транзисторов. Его окончание повлекло за собой смену тренда – производители процессоров стали фокусироваться на многоядерных процессорах для улучшения производительности.

Question 4

В чем заключается идея Bypassing / Data forwarding оптимизации?

Answer 4

Опишем идею bypassing/data forwarding. При конвейерном исполнении инструкций в процессоре могут быть некоторые зависимости по данным или результатам исполнения одной инструкции от другой. Причём, обе эти инструкции могут исполняться одновременно в процессоре, но на разных стадиях конвейера. Без оптимизации придётся ждать, пока данные от ранней инструкции будут записаны по назначенному месту (destination – register or memory) для того, чтобы исполнить позднюю инструкцию. То есть, несколько тактов процессор будет простаивать, что нехорошо для производительности. При использовании bypassing/data forwarding есть провода, соединяющие разные стадии конвейера, чтобы напрямую получить результат операции, а не ждать, пока он будет записан в регистр/память.

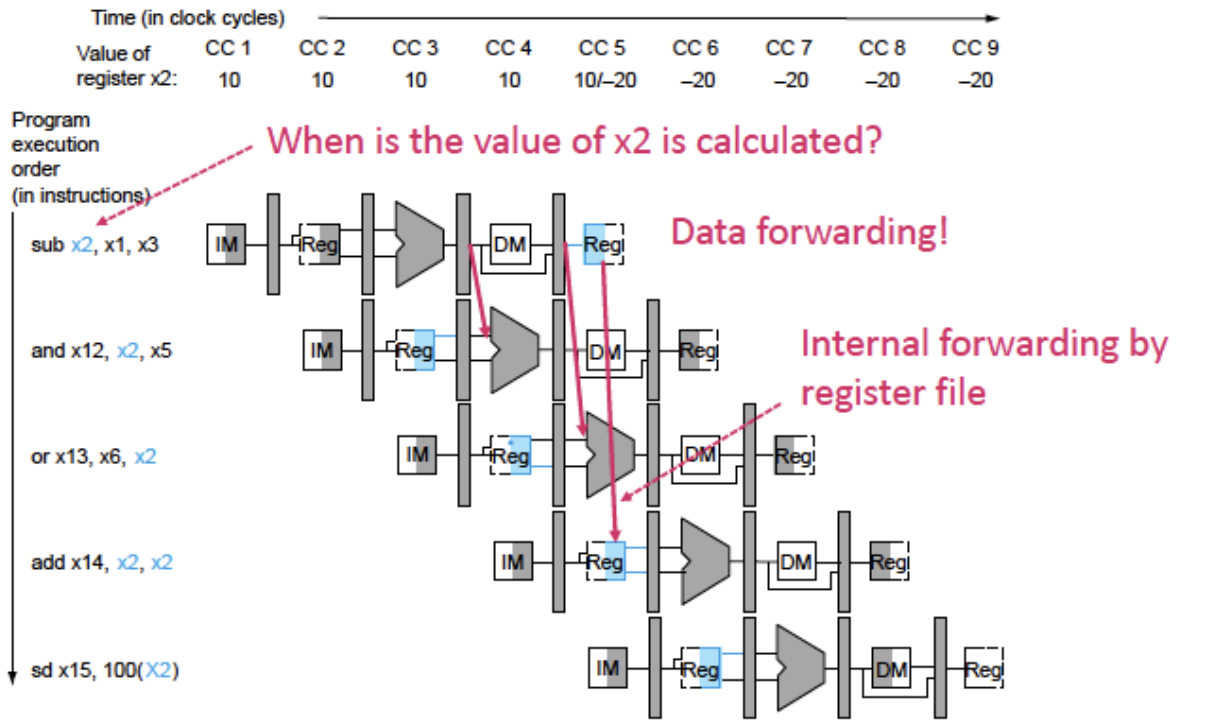

Рис. 1. Bypassing

Question 5

Что такое Instruction-Level Parallelism? Приведите примеры оптимизаций для его повышения

Answer 5

Instruction-Level Parallelism – это процесс одновременного исполнения нескольких инструкций на процессоре. То есть, в один и тот же такт исполняются сразу несколько инструкций. Конвейер процессора – это один из примеров Instruction-Level Parallelism, поскольку в один такт исполняются сразу несколько инструкций, хоть они все и на разных стадиях конвейера.

Примеры оптимизаций для повышения Instruction-Level Parallelism:

- Суперскалярность процессора
- VLIW архитектура (Very Long Instruction Word architecture)
- Векторные CPU, поддержка векторных инструкций

Out-of-Order

Question 1

Для чего нужен Reorder Buffer (ROB)?

Answer 1

В Out-of-Order процессоре инструкции записываются в порядке следования в ROB. Эти инструкции исполняются и в итоге покидают ROB. Но каждая инструкция покидает ROB и обновляет архитектурное состояние процессора только если она самая старая в нём и была исполнена. Таким образом, ROB является некоторым хранилищем инструкций, каждая из которых покидает его только если она и все инструкции до неё были исполнены. Создаётся видимость In-Order исполнения, хотя инструкции, находящиеся внутри ROB, могли исполняться в произвольном порядке. Это позволяет достичь большей производительности.

Question 2

В чем сложность сделать ROB размером 10000 ячеек даже в случае доступной площади и power ресурса?

Answer 2

Существует несколько ограничений, не позволяющих делать (или делающих бесполезным) ROB крайне большого размера:

- Зависимости инструкций: WAR и WAW зависимости крайне неприятны при Out-of-Order исполнении
- Исключения: исключения аппаратуры должны происходить только в

том порядке и тех местах, как если бы инструкции исполнялись In-Order

- Ветвления потока управления: при встрече инструкции, влияющей на поток исполнения, процессору необходимо предугадать, по какой ветви пойдёт исполнение и уже основываясь на этом загружать инструкции в ROB. Если окажется, что предсказание было ложным, то все выполненные инструкции по неверной ветви придётся очистить (flush). Таким образом, теряется производительность

Question 3

Как бороться с False/Anti-Dependencies по регистрам? Что такое Register Alias Table?

Answer 3

Для борьбы с False/Anti-Dependencies аппаратура может содержать больше регистров в спекулятивном состоянии, чем количество архитектурных регистров, и во время исполнения инструкций производить динамический Register Renaming (количество архитектурных регистров не меняется).

Register Alias Table – это таблица, в которой хранится соответствие архитектурных и аппаратурных регистров.

Question 4

Для чего нужна Instruction Queue (IQ) (по другому Scheduler Queue (SQ) или Reservation Station)? Когда инструкции удаляются из ROB, а когда из Scheduler Queue?

Answer 4

Instruction Queue необходима для хранения ещё не исполненных инструкций. Размер IQ значительно меньше размера ROB, что увеличивает производительность, поскольку поиск готовых исполниться инструкций ускоряется (вместо поиска по всему ROB достаточно произвести поиск лишь в IQ). В отличие от ROB, в Instruction Queue хранятся только ещё не исполненные инструкции.

Инструкция удаляется из ROB только тогда, когда она и все предшествующие были исполнены.

Инструкция удаляется из IQ только тогда, когда она была исполнена.

Question 5

Что такое Memory Disambiguation проблема? Как с ней бороться?

Answer 5

Memory Disambiguation – это проблема исполнения инструкций доступа в память в Out-of-Order исполнении. Для борьбы с этой проблемой существует набор правил, который помогает при исполнении Store и Load инструкций.

Для Store инструкций:

- Никогда не исполняются спекулятивно, поскольку произвести откат для данной инструкции невозможно или очень сложно
- Всегда исполняются в том порядке, в каком идут в коде программы (не перемешиваются между собой)

Для Load инструкций:

- Необходимо дождаться, пока все предшествующие Store инструкции были исполнены (или, если иметь дело только с истинными зависимостями, дождаться, пока один конкретный Store будет исполнен)

Для дальнейших улучшений используются:

- Проверка на пересечение по адресу для инструкций Store/Load
- Store Forwarding
- Store Buffer и Load Buffer
- Разбитие Store инструкции на операции вычисления адреса и вычисления данных
- Спекулятивное исполнение Load инструкций

Question 6

Объясните смысл разделения Store инструкции на Store Address/Store Data микрооперации

Answer 6

Для проверки пересечения инструкций Store и Load значение данных не нужно. Адрес в памяти вычисляется, когда операция Store исполняется. Однако инструкция Store отправляется на исполнение только тогда, когда все её входные параметры (адрес и данные) уже готовы. Отсюда вытекает мнимая зависимость от данных (от которых на самом деле нет зависимости).

Для избежания такого рода проблем Store делится на две микрооперации:

- STA – вычисление адреса записи в память
- STD – вычисление данных для записи в память

В этом случае инструкция Load должна ожидать исполнения STD только тогда, когда есть пересечение с соответствующей микрооперацией STA.

Question 7

Что такое Load speculation оптимизация?

Answer 7

Разбиение инструкции Store на две микрооперации STA и STD действительно ускоряет процесс исполнения, но можно добиться ещё лучших результатов. Можно исполнять инструкции Load спекулятивно, не дожидаясь исполнения STA, а потом, после исполнения STA, проверить результат на корректность.

Branch Prediction

Question 1

Какую информацию хранит Branch Target Buffer?

Answer 1

Branch Target Buffer используется для предсказания адреса перехода для инструкций перехода.

По сути – это кэш, хранящий для каждой встреченной инструкции перехода предсказание для её адреса перехода.

Question 2

Опишите, как вы бы представили state-of-the-art предсказатель условных переходов на данный момент

Answer 2

Я бы представил state-of-the-art предсказатель условных переходов на данный момент как чисто PPM-like предсказатель или как гибрид PPM-like и предсказателей, основанных на модели перцептрона. Для меня яркий пример state-of-the-art предсказателя условных переходов на данный момент – это TAGE.

TAGE – это TAgged GEometric history length. Во внимание принимается не только самая длинная последовательность, но и вторая по длине тоже. Небольшой метапредсказатель делает окончательное предсказание на основе всего состояния TAGE. Это сделано для решения проблемы холодного счётчика.

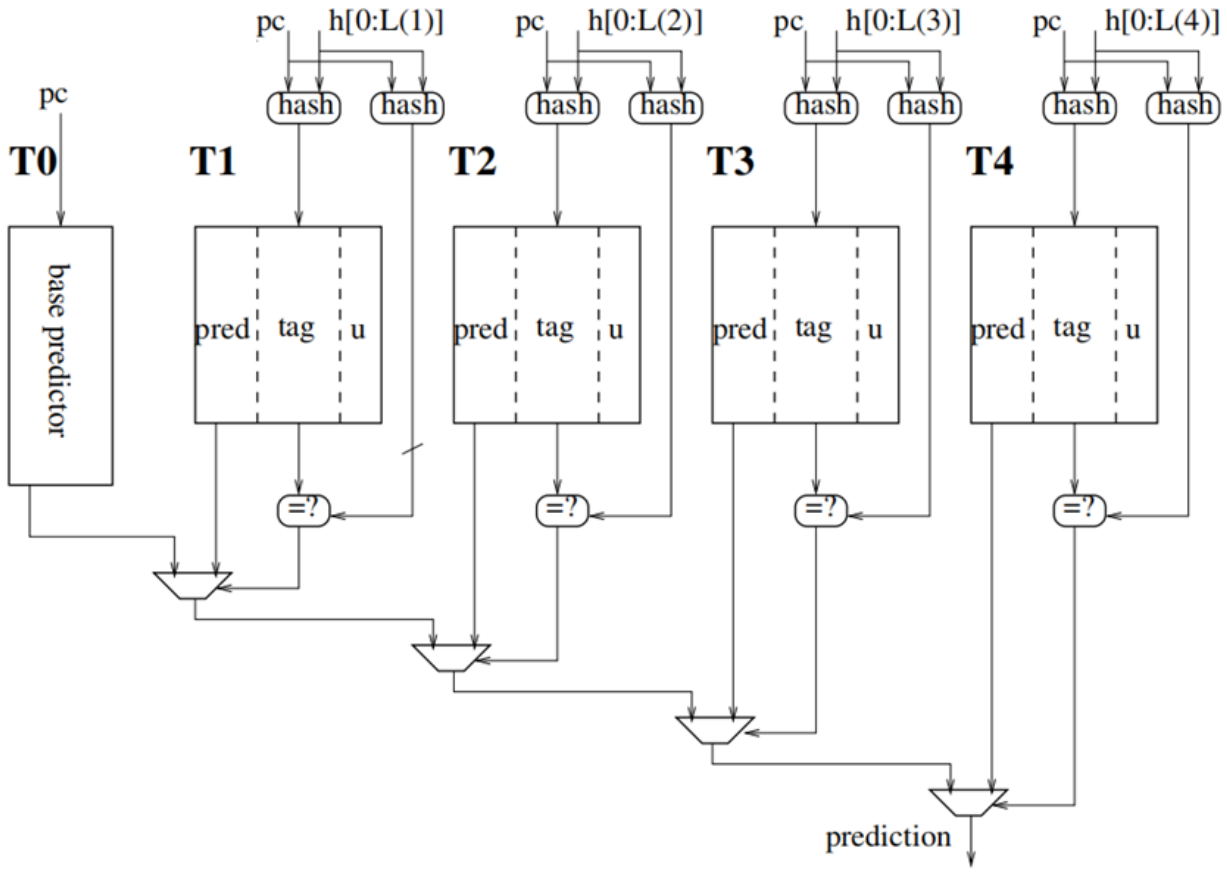

Рис. 2. TAGE предсказатель переходов

Question 3

Процессор имеет двухуровневый адаптивный предсказатель условных переходов (two-level adaptive conditional branch predictor). Предсказатель хранит направления последних бранчей (branch direction) из глобальной истории (global history) в регистре истории (BHR) размером 3 бита. Pattern History Table имеет размер 8 и использует стандартные 2-битные счетчики с насыщением (2-bit saturating counters). В начальных момент времени счетчики находятся в состоянии "Strongly Not Taken". Рассчитайте misprediction rate (процент неверных предсказаний) для данного предсказателя условных переходов в случае исполнения программы ниже

```cpp
#include <iostream>

int main() {
    for (int i = 0; i < 100; ++i) {
        std::cout << i << "\n";
    }
    return 0;
}
```

Answer 3

Ответ неоднозначный. Зависит от нескольких факторов, но главный из них – это компилятор. Реализовать цикл можно двумя разными способами.

Ссылка на Godbolt: https://godbolt.org/z/bnbcbEc7b

Первый вариант (например, Clang):

```
loop_check:
    cmp i, 100
    jge #loop_exit

loop_body:
    ...
    inc i
    jmp #loop_header

loop_exit:
    ...
```

Второй вариант (например, GCC):

```
loop_header:
    jmp #loop_check

loop_body:
    ...
    inc i

loop_check:
    cmp i, 99
    jle #loop_body
    ...
```

Два варианта кардинально отличаются. Разберём их отдельно.

Положим, что бит 0 в BHR соответствует состоянию Not Taken, а бит 1 – Taken. Также положим, что начальное состояние BHR – это три нуля.

Вариант 1

В первом варианте инструкция перехода исполняется 101 раз: 100 раз Not Taken и 1 раз Taken. Учитывая начальные состояния BHR и PHT, то получается, что с самого начала предсказания будут корректными и лишь самый последний раз, при выходе из цикла, предсказание будет ошибочным – в BHR будет 000, а все ячейки PHT будут иметь значение Strongly Not Taken, хотя на самом деле переход будет исполнен.

При таком сценарии количество неверных предсказаний – это 1, а всего их – 101. Получается:

$$Misprediction = \frac{1}{101} \simeq 0.009901 \ (0.9901\%)$$

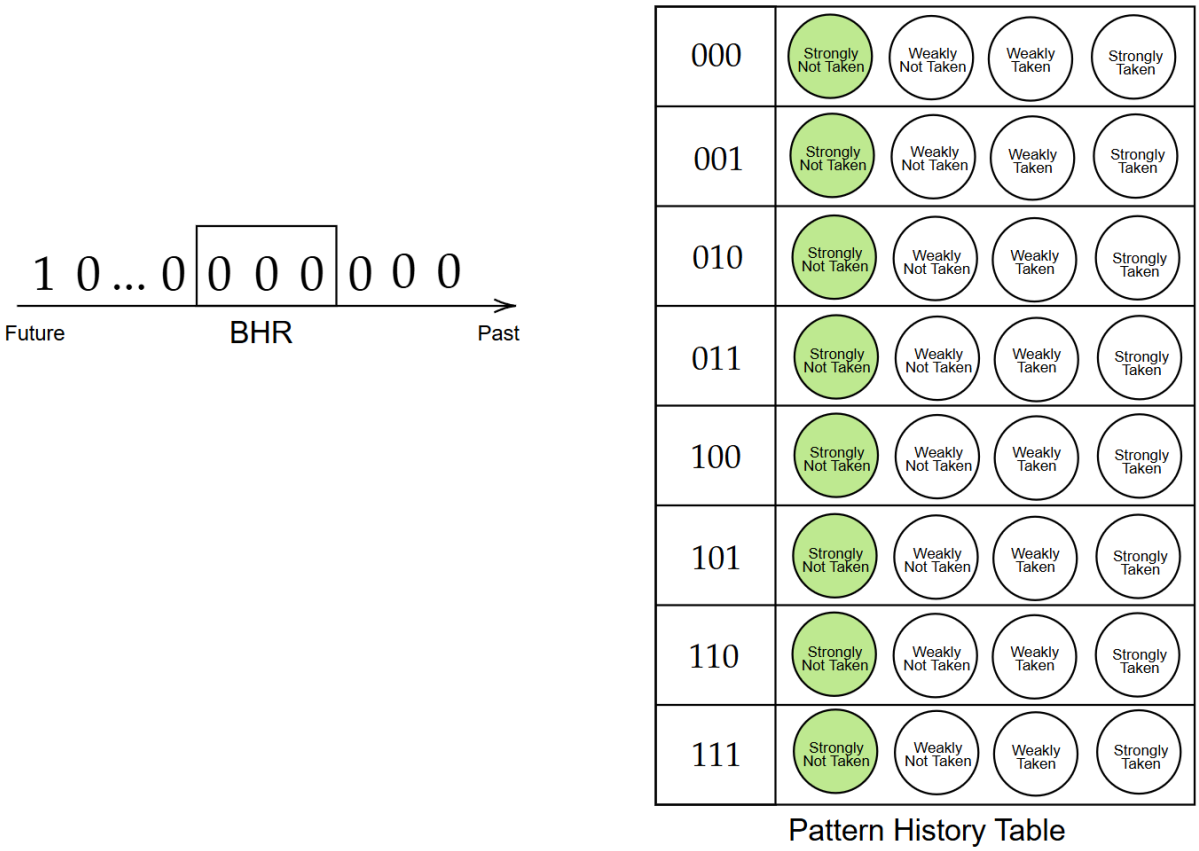

Рис. 3. Вариант 1

Вариант 2

Во втором варианте инструкция перехода исполняется 101 раз: 100 раз Taken и 1 раз Not Taken.

- В первый раз состояние BHR – 000. Соответствующая ячейка в PHT хранит Strongly Not Taken. Это первое неверное предсказание
- Во второй раз состояние BHR – 100. Соответствующая ячейка в PHT хранит Strongly Not Taken. Это второе неверное предсказание
- В третий раз состояние BHR – 110. Соответствующая ячейка в PHT хранит Strongly Not Taken. Это третье неверное предсказание
- В четвертый раз состояние BHR – 111. Соответствующая ячейка в PHT хранит Strongly Not Taken. Это четвертое неверное предсказание
- В пятый раз состояние BHR снова 111. Соответствующая ячейка в PHT хранит Weakly Not Taken. Это пятое неверное предсказание

- Начиная с этого момента, предсказания станут верными, поскольку состояние BHR всегда будет 111, а ячейка PHT – Taken
- ...
- В 101-ый раз состояние BHR – 111. Соответствующая ячейка в PHT хранит Strongly Taken. Это шестое неверное предсказание

При таком сценарии количество неверных предсказаний – это 6, а всего их – 101. Получается:

$$Misprediction = \frac{6}{101} \simeq 0.059406 \ (5.9406\%)$$

Question 4

Объясните связь двухуровневых адаптивных предсказателей (two-level adaptive branch predictor) и алгоритма PPM (Prediction by Partial Matching)

Answer 4

Для наглядности продемонстрируем связь на примере Markov Predictor порядка m.

Оба алгоритма рассматривают последние m бит входной последовательности. Но в двухуровневых адаптивных предсказателях для каждой такой подпоследовательности используются 2-битные счётчики с насыщением, а у Markov Predictor – обыкновенные счётчики частоты появления. Окончательное предсказание делается тоже по-разному: в двухуровневых адаптивных предсказателях – по положительному/отрицательному значению счётчика (то есть к какой его границе ближе); а в Markov Predictor – выбирается на основе максимальной частоты появления.

Оба алгоритма хорошо динамически подстраиваются под поведение программы для предсказания переходов.

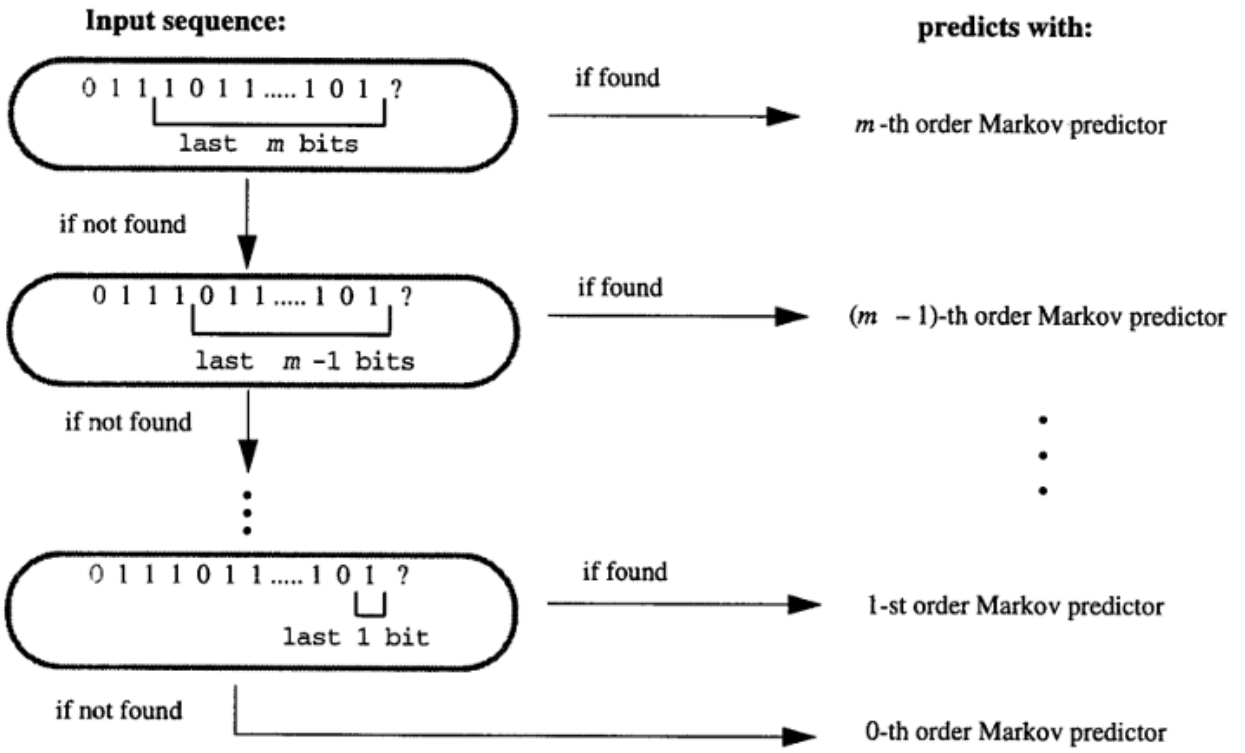

Рис. 4. Prediction by Partial Matching

	a two-level predictor with m bit history	a Markov predictor of order m
First level (same)	0 1 1 1 0 1 1 0 ... 1 0 1 ? last m bits search for a match in the last m bits (same for both predictors)	
Second level (a majority vote)	one 2-bit counter positive 00 01 +1 taken negative 10 11 -1 not taken	2 counters frequency counter for 1's frequency counter for 0's
Decision based on: (the majority)	positive or negative count	max(0's frequency, 1's frequency)

Рис. 5. Two-level adaptive branch predictor vs PPM

Question 5

Какими полезными свойствами обладает перцептрон (perceptron) с точки зрения обработки входных данных? Опишите, какую функцию выпол-

няет перцептрон в случае использования в виде статистического корректора (statistical corrector)

Answer 5

Предсказатели на основе модели перцептрона очень хороши при комбинировании входных данных разных типов.

Предсказатель переходов TAGE крайне эффективен при предсказании коррелированных переходов. Однако он плохо предсказывает переходы, которые статистически более связаны с направлением перехода, а не с их историей, или вообще некоррелированны. Для лучших предсказаний таких переходов был предложен Statistical Corrector – небольшой предсказатель, который нацелен на детектирование маловероятного предсказания от самого TAGE и инвертирование его. Для этого ему необходима информация об инструкции перехода: адрес, глобальная история переходов, глобальный путь управления, локальная история и т.п.

Question 6

Какие виды условных переходов являются сложно-предсказываемыми для предсказателя TAGE?

Answer 6

Для TAGE слабо коррелированные или совсем некоррелированные переходы (weakly-biased branches) являются сложно-предсказываемыми. Именно поэтому и был предложен Statistical Corrector.

Question 7

Объясните основный смысл рассмотренной на лекции статьи "Branch Prediction is Not a Solved Problem"

Answer 7

Несмотря на то, что в мире уже существует огромное множество разных предсказателей переходов, которые дают ощутимый прирост производительности, проблема предсказания переходов остаётся актуальной, потому что по оценкам за оставшимися процентами неверных предсказаний (misprediction rate) скрывается ещё около 20% производительности (IPC). Главные причины этого: существование сложно-предсказываемых переходов и наличие переходов, которые в динамике крайне редко исполняются, поэтому они вносят вклад в misprediction rate.

Memory

Question 1

В чем разница между Fully Associative, Direct Mapped и Set-Associative кэшами?

Answer 1

Fully Associative:

- Кэш-линии может быть сопоставлена произвольная ячейка кэша. Поэтому поиск происходит по всему кэшу
- Адрес разбивается на номер линий (тег) и смещение внутри неё. Поэтому коллизии невозможны
- Каждая ячейка кэша имеет тег. Поэтому номер линии адреса сравнивается со всеми ячейками кэша в параллель

Direct Mapped

- Кэш-линии может быть сопоставлена лишь определенная ячейка кэша. Поэтому для поиска необходима только одна ячейка
- Адрес разбивается на номер линий (тег), номер ячейки кэша (сет) и смещение внутри неё. Поэтому коллизии возможны

N-Way Set-Associative

- Кэш-линии может быть сопоставлена лишь определенный набор ячеек кэша (набор ways для каждого сета)
- Адрес также разбивается на номер линий (тег), номер ячейки кэша (сет) и смещение внутри неё. Поэтому коллизии возможны, но происходят сильно реже, чем в Direct Mapped кэше

Question 2

Для чего используется Translation Lookaside Buffer (TLB)?

Answer 2

Для безопасности (и иногда эффективности) программы работают не с физической памятью, а с виртуальной. Процессору необходим настоящий, физический, адрес для операций с памятью. За трансляцию адресов ответственны MMU (Memory Management Unit) и операционная система. Но это достаточно долгий процесс, поэтому существует (Translation Lookaside Buffer) TLB – это небольшой кэш для транслированных адресов для ускорения этого процесса.

Question 3

Что такое гранулярность политик замещения?

Answer 3

Гранулярность политик замещения отвечает за то, по каким критериям и на каких стадиях выделять ячейки кэша. Существуют крупные, или крупнозернистые (coarse-grained), и тонкие, или мелкозернистые (fine-grained), политики замещения. В первых кэш-линии различаются лишь на поведении в кэше, а вторые – ещё вдобавок при вставке в кэш.

Question 4

Какие есть стандартные паттерны обращения к кэшу?

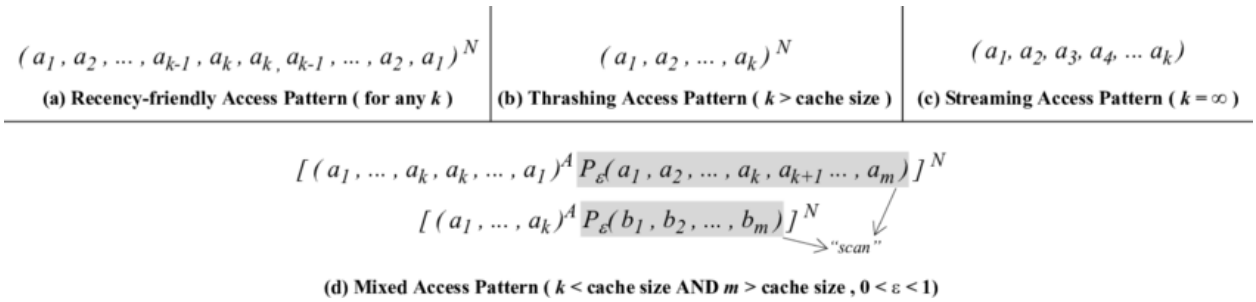

Рис. 6. Стандартные паттерны обращения к кэшу

Answer 4

Паттерны обращения к кэшу:

- Recency-friendly
- Thrashing
- Streaming
- Mixed (Scanning)

Question 5

В каком ключе LRU Insertion Policy (LIP) лучше классической LRU политики замещения?

Answer 5

LRU Insertion политика очень похожа на классическую LRU. Отличие только в том, что при вставке в кэш в LIP выбирается LRU позиция (самая давняя), а классическом LRU – MRU позиция (самая новая). Благодаря этому LIP более устойчив паттерну обращения Thrashing.

Question 6

За счет чего BRRIP политика замещения является устойчивой к scanning и thrashing паттернам?

Answer 6

Политика замещения BRRIP устойчива к паттерну обращения к кэшу thrashing ровно по той же причине, что и LIP – если к недавно вставленной линии не было обращения, то она быстро вытесняется из кэша.

Политика замещения BRRIP устойчива к паттерну обращения к кэшу scanning за счёт того, что вставка новых линий происходит ближе к позиции жертвы (victim), причём, с некоторой вероятностью новоявленная линия будет вставлена не непосредственно на позицию жертвы, а лишь перед ней, что позволит ей задержаться в кэше подольше.

Question 7

В чем основная идея политик замещения Hawkeye и Mockingjay?

Answer 7

Главная идея политик замещения Hawkeye и Mockingjay заключается в том, что, несмотря на то, что предсказать будущее невозможно, можно применить известный алгоритм Белади (Belady) к доступам в память в прошлом. Таким образом, можно обучить предсказатель для того, чтобы предсказывать будущие доступы в память.

Question 8

Объясните принцип работы Spatial Memory Streaming (SMS) префетчера

Answer 8

Spatial Memory Streaming (SMS) разделяет адресное пространство на секции фиксированного размера, которые называются Spatial Regions. SMS

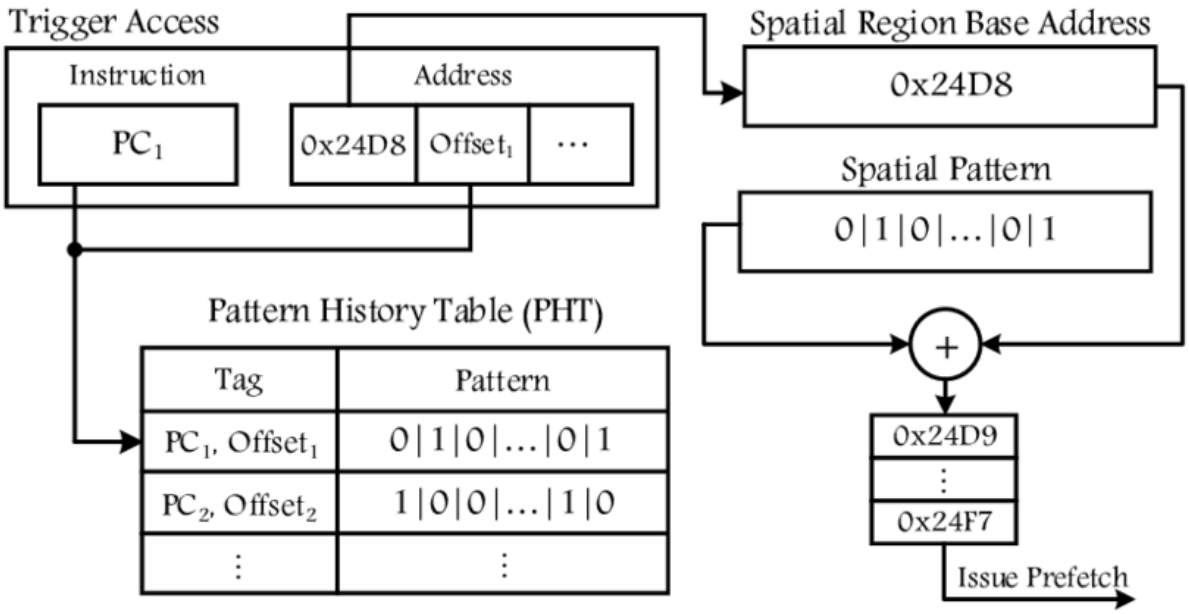

Рис. 7. Spatial Memory Streaming префетчер

запоминает паттерны доступа в память в этих секциях, после чего эти паттерны используются для префетчинга будущих доступов в память, если приложение снова обращается к этим или похожим секциям.

Advanced Optimizations & Parallelism

Question 1

На покрытие какие сложных случаев направлено использование Execution-based Prefetchers?

Answer 1

На покрытие случаев, когда в программе происходит множество обращений в память, которые могут повлечь множество промахов в кэш, из-за чего конвейер процессора будет простаивать (stall).

Question 2

Что такое Value Prediction оптимизация? Для каких инструкций, как правило, используется данная оптимизация в современных процессорах?

Answer 2

Value Prediction оптимизация заключается в предугадывании результатов инструкций до их непосредственного исполнения. Оптимизация направлена на устранение зависимости по данным и увеличения Instruction-Level Parallelism.

Данная оптимизация, как правило используется для:

- Арифметических инструкций
- Load инструкций
- Инструкций, которые могут дать ощутимый прирост при малой стоимости ошибки, если вероятность предсказания мала

- Инструкций, которые могут дать малый прирост при ощутимой стоимости ошибки, если вероятность предсказания велика

Question 3

За счет чего получается прирост производительности в случае применения Value Prediction оптимизации?

Answer 3

За счёт устранения некоторых зависимостей по данным и за счёт наличия Value Locality.

Question 4

В чем разница между Value Prediction и Branch Prediction с точки зрения необходимости/ожидаемого положительного эффекта от предсказания и последующего спекулятивного исполнения в случае низкой уверенности в точности предсказания?

Answer 4

Оптимизация Branch Prediction предсказывает поток управления программы, а Value Prediction – непосредственно значения инструкций, что гораздо сложнее. Стоимость ошибки предсказания Branch Prediction мала по сравнению с потенциальным выигрышем при успехе, поэтому данная оптимизация может исполняться спекулятивно всегда. Value Prediction – напротив – имеет высокую стоимость ошибки предсказания, но также и более крупный выигрыш при успехе, поэтому для данной оптимизации необходима хорошая уверенность в точности предсказания.

Question 5

Что такое Memory Renaming оптимизация?

Answer 5

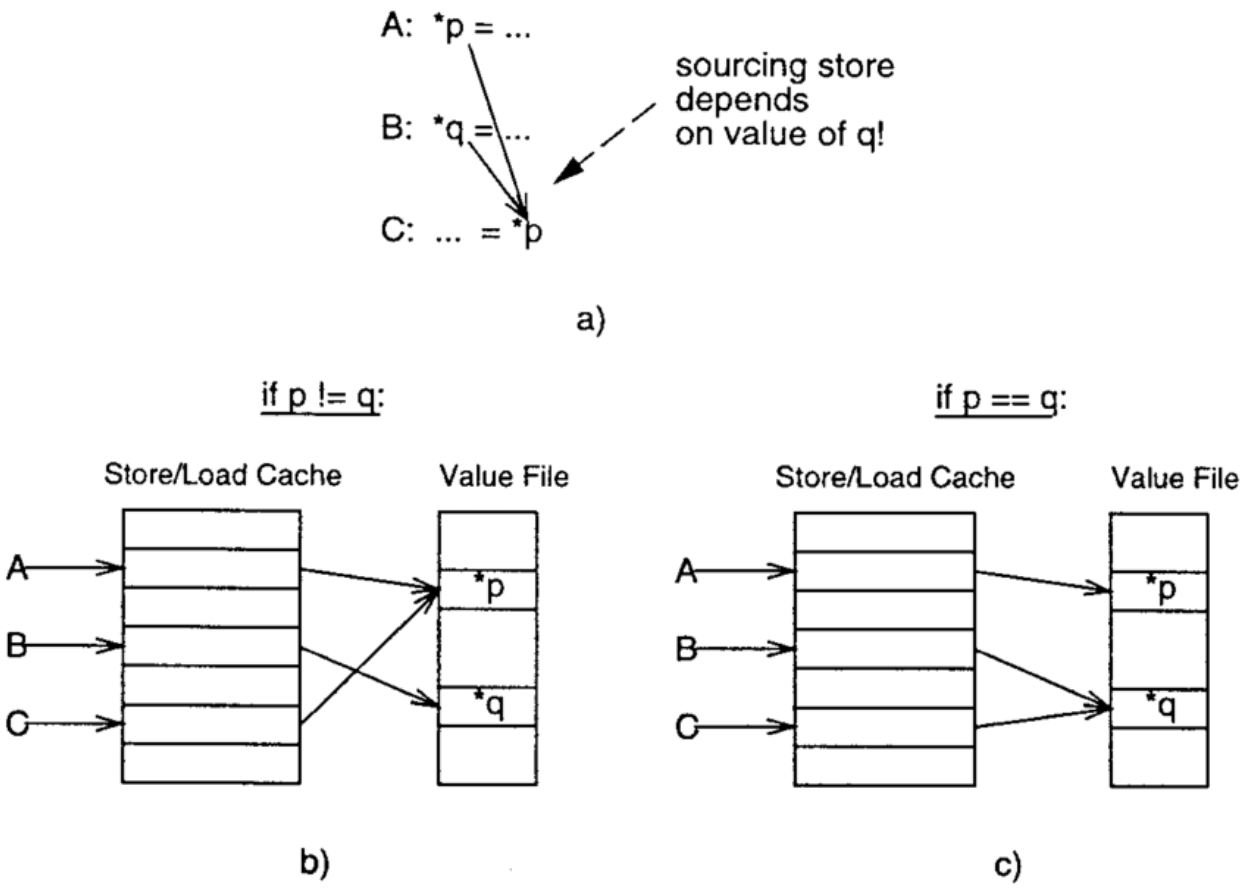

Рис. 8. Memory Renaming оптимизация

Memory Renaming – это оптимизация, направленная на улучшение коммуникации инструкций load/store с памятью, которая находит, кэширует и использует зависимости между load/store инструкциями. Как только определенная зависимость между load/store инструкциями была найдена, записи в память с помощью инструкции store могут быть перенаправлены напрямую к зависимым от них инструкциям load. Таким образом, улучшается коммуникация инструкций load/store с памятью и, следовательно, производительность.

Оптимизация вносит некоторые дополнительные правила работы с инструкциями load/store.

Для инструкции Store:

```
if (not in store/load cache) {
    allocate store/load cache entry for store
    allocate value file entry for store result
    point store/load cache entry to value file entry
}
deposit store result into value file and memory
```

Для инструкции Load:

```
if (not in store/load cache) {
    allocate store/load cache entry for load
    point store/load cache entry to value file entry of source store
    if no sourcing store, insert result of load into value file
}
return value file entry as load result
```

Question 6

Объясните разницу между Fine-Grained Multithreading, Coarse-Grained Multithreading и Simultaneous Multithreading

Answer 6

Fine-Grained Multithreading: смена потока на каждом такте процессора – в конвейере нет двух или более инструкций с одного и того же потока.

Coarse-Grained Multithreading: смена потока при каком-то определенном событии, например, промахе в кэш, синхронизации или по истечении кванта времени.

Simultaneous Multithreading: исполнение инструкций сразу с нескольких потоков в один и тот же такт процессора для большей загрузки его элементов.

Fine-Grained Multithreading:

- Нет нужды для проверки зависимостей между инструкциями
- Нет нужды в логике предсказателя переходов (branch predictor)
- Пузыри-такты (bubble cycles) используются другими потоками для исполнения инструкций
- Улучшенные Throughput, Latency Tolerance и Utilization
- Дополнительная сложная логика в аппаратуре
- Хуже производительность одного конкретного потока
- Борьба потоков за ресурсы памяти и кэша
- Всё ещё необходимы проверки логики исполнения между потоками

Coarse-Grained Multithreading:

- Улучшенная производительность одного конкретного потока
- Сложнее логика в аппаратуре
- Бóльшая нагрузка при смене контекста
- Fairness vs. Throughput проблема

Simultaneous Multithreading:

- Бóльшая загрузка элементов процессора
- Суперскалярные и Out-of-Order процессоры уже имеют поддержку нескольких функциональных устройств
- Лучший Throughput

Thank you!