

# Spectre & Meltdown

Борисав Живановић

11. фебруар 2023.

- 1 Архитектура и микроархитектура
- 2 Кеширање
- 3 Предвиђање гранања и прекоредно извршавање
- 4 Основни механизми изолације
- 5 Употреба кеша као side-channel
- 6 Spectre
- 7 Meltdown

# Шта рачунар заиста зна да ради?

- Језик рачунара: **скуп инструкција** (енгл. ISA, Instruction Set Architecture)
- Аритметичке операције: **add, sub, div, mul, ...**
- Померање података:
  - са улазног уређаја у меморију
  - из меморије на излазни уређај
  - са једне меморијске локације на другу
- Условно гранање: извршавање кода уколико је логички услов испуњен

# Условно гранање

- Кључни механизам - омогућава имплементацију било ког алгоритма
- Концпети виших програмских језика као што су **if**, **else**, **for**, **while**, **switch** се свODE на условно гранање

# Instruction Set Architecture

- Представља слој апстракције изнад микроархитектуре
- Главна разлика између различитих ISA је у количини логики коју појединачна инструкција може да садржи
- Подела: CISC (Complex Instruction Set Computer), RISC (Reduced Instruction Set Computer)
- x86 је иницијално био класична CISC архитектура
- Данас x86 инструкције на нивоу ISA се превode у **микроинструкције** налик на RISC инструкције
  - ово се дешава на нивоу микроархитектуре и програмер тога није свестан!

# Микроархитектура

- Представља конкретну имплементацију ISA
- Замисао је да се кроз време имплементација побољшава, а да се задржи компатибилност са постојећим софтвером
- Неке разлике између различитих микроархитектура:
  - параметри кеша (капацитет, асоцијативност, величина линије, број нивоа)
  - број језгара
  - подршка за механизме који крше секвенцијалан модел извршавања

# Меморијска хијерархија I

*Ideally one would desire an indefinitely large memory capacity such that any particular... word would be immediately available... We are... forced to recognize the possibility of constructing a hierarchy of memories each of which has greater capacity than the preceding but which is less quickly accessible.*

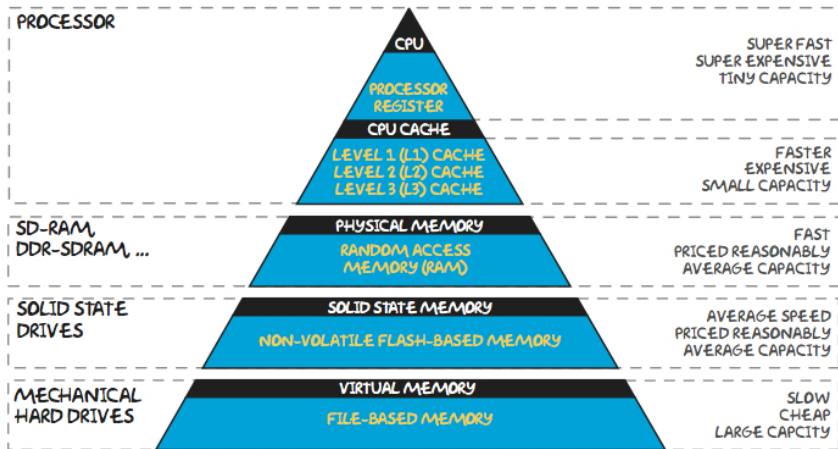
**Burks, Goldstine, von Neumann (1946)**

# Меморијска хијерархија II

- Проблем: не постоји бесконачно брза и бесконачно велика меморија
- Чињеница: постоје технологије меморије које омогућавају релативно велики капацитет, по цену релативно мале брзине
  - ...као и обрнуто!
  - брзина и капацитет меморије су, по правилу, обрнуто сразмерни
- Да ли је могуће добити највећи капацитет уз највећу брзину, по најмањој цени?
- Меморијска хијерархија нам ово *донекле* омогућава
  - цена: *приближно* најспорија меморија
  - брзина: *приближно* најбржа меморија



# Меморијска хијерархија III



# Локалитет I

- **Просторни локалитет:** уколико је некој локацији приступљено, вероватно ће бити приступљено и суседним локацијама
  - пример: приступање суседним елементима низа, извршавање наредних инструкција
- **Временски локалитет:** уколико је некој локацији приступљено, вероватно ће јој бити приступљено у скоријем временском периоду
  - пример: позив методе у петљи, приступ елементима *linked list*
- Мерењима је доказано да програми поштују наведене особине

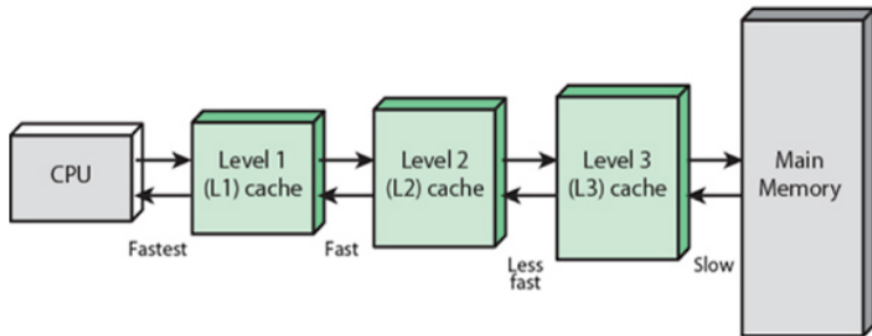
## Локалитет II

- Цео меморијски подсистем је оптимизован за програме који поштују локалитет
  - уколико покренемо заједно један програм који поштује локалитет (Matlab) са другим који не поштује (GCC), може да дође до давања предности оном који поштује!
- Поједине специјализоване архитектуре избацују кеш меморију уколико није могуће направити решење које поштује локалитет
  - у овом случају, кеширање би успорило програм
- Занимљивост: постоје случајеви у којима су операције над *array list* брже него над *linked list*, јер иако је временска сложеност операција већа, операције се далеко брже извршавају уколико је цео низ у кешу!

# Кеш меморија I

- Налази се у процесору, најчешће имплементирана у SRAM технологији
- Чува тренутно потребан подкуп радне меморије програма
  - подкуп се динамички одређује уз претпоставку локалитета
- Програмер не мора да буде свестан конкретне имплементације кеша
  - али је то, у одређеним случајевима, пожељно
  - постоје инструкције које омогућавају измену стања кеша (CLFLUSH, PREFETCHW)
- Сваки меморијски приступ мора да прође кроз кеш

# Кеш меморија II



# Кеш меморија III

- **Cache hit:** тражени податак је пронађен у кешу
- **Cache miss:** тражени податак није пронађен у кешу
- **Hit time:** време које је потребно да се утврди да ли је тражени податак у кешу
- **Miss time:** време које је потребно да се тражени податак добава у кеш

# Предвиђање гранања

- `if(x < y) {...} else {...}`
- Променљиве `x` и `y` представљају вредности из радне меморије
- Одређивање гране коју треба извршити није могуће док обе вредности не буду добављене у кеш
- Условно гранање често изазива **cache miss**
  - ...и на тај начин зауставља рад процесора
- Идеја:
  - извршавање гране за коју се претпоставља да ће бити изабрана док се чека IO
  - чување или одбацивање резултата након утврђивања да ли је извршавање гране требало да се деси

# Прекоредно извршавање

- $a1 = b1 + c1;$   
 $a2 = b2 + c2;$   
 $a = a1 + a2;$
- Шта ако су вредности **b2** и **c2** у кешу, а потребно је прво сачекати добављање **b1** и **c1**?
- У секвенцијалном моделу извршавања, процесор би био заустављен
- Идеја:
  - на нивоу ISA задржавамо секвенцијални модел извршавања
  - на нивоу микроархитектуре инструкције извршавамо у редоследу који зависи од доступних података
  - резултат програма мора остати исти!



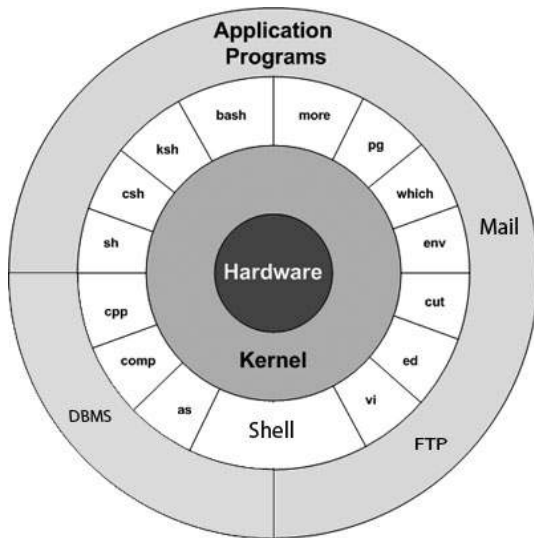
# Индирекција

- Било који проблем у рачунарству може бити решен још једним нивоом индирекције, осим наравно проблема превише индирекција (David J. Wheeler)
- Индирекција омогућава имплементацију контроле приступа
- Извршавање **акције** мора да одобри **посредник** који дефинише правила приступа

# Контрола приступа у хардверу I

- Рачунар без контроле приступа би донекле био употребљив у једнокорисничком окружењу
  - ...али неупотребљив у вишекорисничком
  - чак и у једнокорисничком окружењу, одсуство изолације процеса представља велику опасност
- Основне градивне блокове је неопходно имплементирати у хардверу
  - софтвер можда неће бити рад да сарађује!
- Кључни механизми: режими рада процесора, виртуелна меморија
- Додатне контроле се имплементирају у кернелу
  - пријављивање корисника на систем, пермисије за приступ фајловима

# Контрола приступа у хардверу II



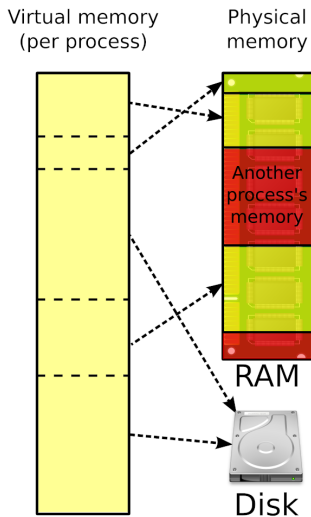
# Режими рада процесора

- Привилеговани: IO, меморијске табеле, табеле прекида
  - кернел
- Неривилеговани: аритметичко/логичке операције, условно гранање, ограничен приступ меморији, системски позив
  - кориснички софтвер
- Прелазак из непривилегованог у привилеговани режим је могућ приликом прекида или системског позива
- Кернел одбија захтев уколико кориснички процес нема потребне привилегије и убија га

# Виртуелна меморија I



# Виртуелна меморија II



# Покретање оперативног система

- Процесор се буди у привилегованом режиму
- Учитава се кернел
- Иницијализују се табеле прекида
- Иницијализују се меморијске табеле
- Контрола се предаје корисничким програмима, прелази се у непривилегован режим
- Овако подешен посредник више није могуће уклонити или заобићи
  - ...под претпоставком да нема багова у имплементацији кернела и хардвера

# Употреба кеша као side-channel I

- До сада смо причали о механизмима који нам омогућавају безбедно дељење рачунара и побољшање перформанси секвенцијалног модела
- Безбедност и перформансе су често супротни захтеви у дизајну!
- Заједничко за оба напада је да извршавају недозвољен меморијски приступ **спекулативно**
  - пошто је извршавање **спекулативно**, процесор ефекте чува/одбацује накнадно
  - комплетно одбацавање резултата није могуће
  - стање ISA се успешно поништава, али остаје видљиво стање микроархитектуре (вредности у кешу!)



## Употреба кеша као side-channel II

- Код спекулативног извршавања процесор не зна да ли да баци грешку за недозвољен меморијски приступ
- Приступ вредности уčitаној у кеш је видно брже од вредности из радне меморије
- Алгоритам:
  - алоцирамо низ
  - изазивамо недозвољен приступ меморији
  - приступамо елементу низа чији индекс одговара тајној вредности
  - итерирамо кроз низ и меримо време потребно за приступ индексима
  - уклањамо цео низ из кеша и приступамо наредној недозвољеној адреси

## Употреба кеша као side-channel III

елемент	кодирана вредност	време приступа
$a[x * 0]$	0	300 ns
$a[x * 1]$	1	300 ns
$a[x * 2]$	2	5 ns
$a[x * 3]$	3	300 ns
...	...	...
$a[x * 254]$	254	300 ns

Скалирање за вредност  $x$  је неопходно јер би у супротном једна линија кеша представљала 16 вредности, јер је `int` ширине 32 бита, а величина линије кеша је углавном 64 бајта

# Spectre: увод

- Мета напада: кориснички део меморије активног процеса
- Због чега ово представља проблем?
  - процес би требало да види своју корисничку меморију
- Проблем: Web/JavaScript
  - све скрипте се извршавају у оквиру истог процеса
  - у оквиру истог адресног простора се налазе тајне вредности различитих веб апликација
  - интерпретер онемогућава недозвољен приступ
  - ...али нема начин да спречи спекулативно извршавање

# Spectre: напад I

```
if(x < arr_size) {  
    y = side_chan_arr[arr[x] * 4096];  
}
```

**x**: вредност коју бира нападач

**y**: додела вредности која се успешно одбацује

**arr**: помоћни низ који користимо како би приступили недозвољеној адреси

**side\_chan\_arr**: низ који користимо као side-channel. присуство елемента низа у кешу се не одбацује!

## Spectre: напад II

```
if(x < arr_size) {  
    y = side_chan_arr[arr[x] * 4096];  
}
```

Напад почињемо тако што прво као  $x$  бирамо валидну вредност ( $x < arr\_size$ ). Овиме *убеђујемо* предиктор гранања да је грану углавном потребно извршити.

## Spectre: напад III

```
if(x < arr_size) {  
    y = side_chan_arr[arr[x] * 4096];  
}
```

Потом, бирамо вредност  $x$  која излази из граница низа. Добијена адреса представља потенцијално скривену вредност из адресног простора нашег процеса.

## Spectre: напад IV

```
if(x < arr_size) {  
    y = side_chan_arr[arr[x] * 4096];  
}
```

Уколико је предиктор гранања одлучио да изврши грану, долази до приступа елементу из `side_chan_arr` који одговара добијеној вредности.

Ово изазива довлачење дела низа `side_chan_arr` у кеш.

## Spectre: напад V

```
if (x < arr_size) {  
    y = side_chan_arr[arr[x] * 4096];  
}
```

Процесор накнадно закључује да извршавање гране није требало да се деси јер се `x < arr_size` евалуира у `false`.

Додела вредности променљивој `y` се успешно одбацује.

Стање кеша се не одбацује! Линија је добављена и њено присуство је могуће проверити.



## Spectre: напад VI

```
for(i = 0; i < pow(2, 32); i++) {  
    t1 = time();  
    y = side_chan_arr[i * 4096];  
    t2 = time();  
  
    elapsed = t2 - t1;  
  
    if(elapsed < 5) {  
        println("Stolen_value: ", i);  
    }  
}
```

## Spectre: напад VII

Итерирамо кроз `side_chan_arr` и меримо време потребно за приступ елементима низа.

Уколико је елемент присутан у кешу, успешно смо преко side-channel пренели вредност.

Приказани алгоритам је упрошћен јер не узима у обзир случајеве у којима више елемената проналазимо у кешу, или не проналазимо ниједан.

# Литература

- Spectre Attacks: Exploiting Speculative Execution
- Meltdown: Reading Kernel Memory from User Space
- Computer Organization and Design: The Hardware/Software Interface, David A. Patterson & John L. Hennessy
- Системски софтвер (презентације), Иван Хејгебауер
- Operating Systems: Internals and Design Principles, William Stallings
- Preliminary Discussion of the Logical Design of an Electronic Computing Instrument