Year:..... Month:..... Day:.....

«جلسه اول»

اسلاید ۶:

معماری کامپیوتر یعنی طراحی abstraction layer یا سطح تجرید متفاوت و متنوعی که به ما این امکان را میدهد که پیاده سازی کنیم اپلیکیشن های information processing به صورت کارا و درست با کمترین هزینه و بهترین بهره وری با تکنولوژی ساخت که امروزه قرار دارد (معماری کامپیوتر در ساخت تکنولوژی دخالت ندارد ولی از تکنولوژی در دسترس باید بهترین استفاده کند که به اپلیکیشن مورد نظر برسد)

اسلاید ۷:

Gap (فاصله) زیادی بین physical و Application است که این اسلاید این Gap را نشان می دهد

این اسلاید abstraction layer هایی که معمار باید طراحی کند را نشان می دهد

ما یک Application داریم که یک کار مشخصی دارد که همین Application می خواد طراحی شود باید الگوریتم آن نوشته و طراحی شود بعد از طراحی الگوریتم باید آن را implement کنیم در یک زبان برنامه نویسی که این زبان برنامه نویسی باید در یک بستری مثل OS و linux و ... نوشته شود در نهایت برنامه ما از نظر نرم افزاری ساخته می شود حالا با استفاده از کامپایلر کار کامپایل برنامه را انجام می دهد کامپایلر زبان قابل فهم و سطح بالای ما را تبدیل می کند به زبان سطح ماشین. این زبان ماشین و دستورالعمل ها بر یک مبنایی مثل ترمه اینجا میشه سطح پایین تر که ساختار داخلی پردازنده است که میشه سطح microarchitecture (ریزمعماری) که می شود المان های داخلی یک پردازنده مثلاً Elu این شبکه، واحد ضرب کننده و اینجا Cache است و ... که این میشه معماری داخلی یک پردازنده که برمبنای اون ۵ مراحل بالا صورت می گیره Application ساخته شده. microarchitecture

اون مدارهای داخلی پردازنده است که اون مدارها در سطح پایین تر در RTL طراحی شدند مثلاً ورودی های رجیستر ها از فلان جا بیاد و در فلان جا ذخیره شود. خود این نقل و انتقال لایه پایین تر دارد که می شه Circuits.

در سطح Circuits مثلاً یک رجیستر داریم شکل داخلیش AND و OR و ... داشته باشه (یعنی یک شکل داخلی چجوریه و از چه چیزهایی تشکیل شده) جدول کارنو داشته باشه و ... تا مدارهای پایه را بسازیم

سطح پایین تر آنچه المان های پایه ای مدار چیه، ترانزیستورها، خازن ها، مقاومت ها، سیم ها ..... که ابتدایی ترین

Scanned by CamScanner

Year:..... Month:..... Day:.....

مدارهای جامع مثل AND و OR و ... را می‌سازند و سطح پایین‌تر یعنی physical که دیگه کاری به بینایی کامپیوتر ندارد
معمار کامپیوتر باید Application را وصل می‌کند به pc

اسلاید ۸:

تکنولوژی‌ها هستند که بستر برای Application شکل می‌گیره ولی لزوماً این قول نیست که همه Application ها هستند
که تکنولوژی را به سمت جلو حرکت می‌دهند تا تکنولوژی‌های جدیدی را بسازد
در این تحولات باید حواسمون به Compatibility باشه Compatibility یعنی هر چیز جدیدی که می‌سازیم با
نسل گذشته و هر چیزی که تا الان بوده سازگار باشد

اسلاید ۹:

در این اسلاید روند تحولات در دنیای کامپیوتر ۱۰۰، ۲۰ سال اخیر را نشان می‌دهد. محور y قدرت پردازش کامپیوترهاست
و محور x زمان است و نمودار تکنولوژی‌های مختلف این ۱۰۰ سال را نشان می‌دهد در اوایل ۱۹۰۰ کامپیوترها الکترومکانیکی
بودند یعنی هم قطعات الکترونیکی داشتند و هم مکانیکی در ۱۹۲۰ Relay ها اومدند که کار یک کلید را انجام می‌دهد
در ۱۹۵۰ لامپ خلأ اومدند در ۱۹۶۰ و ۱۹۷۰ ترانزیستورها اومدند که انقلاب اصلی کامپیوتر در دهه ۷۰ به وجود
آمد که مدارها (IC ها) به وجود آمدند که چندین هزار ترانزیستور را می‌توانند به هم وصل کنند از اون به بعد دیگه تحول
چنانی به وجود نیامد

اسلاید ۱۱:

از سال ۱۹۵۰ تا ۳۰، ۴۰ سال بعد آن چیزی که دنیای کامپیوتر را متحول می‌کرد فقط تکنولوژی بود یعنی قانون moore
بود که هر سال کامپیوترها سریع‌تر و ارزان‌تر و کوچک‌تر یا توان مصرفی کمتر می‌ساخت
حدود ۵ سال اخیر تکنولوژی ساخت به یک سد بزرگ خورد یکی از این سد بزرگ این بود که Dennard scaling به پایان رسید
Dennard گفته با پیشرفت تکنولوژی در دنیای IC ، power density ثابت می‌مونه یعنی در واحد سطح
power که مصرف می‌کنیم از این تکنولوژی به اون تکنولوژی ثابت می‌مونه و moore گفته بود هر دو سال تعداد ترانزیستور در
واحد سطح دو برابر می‌شود اگر دو برابر بشود آن وقت در مساحت مشخص ترانزیستور بیشتری داریم، قدرت پردازش
بیشتری داریم و performance بیشتری می‌بریم ، Dennard می‌گه ترانزیستور دو برابر شدن مصرف انرژی را

Scanned by CamScanner

Year:..... Month:..... Day:.....

تبدیل می‌کند و مصرف انرژی همچنان ثابت است در این صورت طبق Dennard مصرف انرژی از این تکنولوژی به اول کمی
را می‌توانیم کاهش بدهیم اما ولتاژ از یک مقداری پایین‌تر نمی‌تونه بیاد (مقدار Threshold) بنابراین Dennard دیگه جواب نمیده
یه دیگه از این سد بزرگ اینکه سلول قانون مور به انتها رسیده و دیگه نمی‌تونیم ترانزیستورها را از یک حدی کوچک‌تر کنیم (سرعت بود
ترانزیستورها هر دو سال دو برابر می‌شد یعنی ابعاد کوچک می‌شد که تعداد زیاد بشه)

مشکل دیگر درباره اینکه چیزی نداریم جایگزین کنیم حالا که Dennard و moore به مشکل خوردن و دنیای کامپیوتر
داره به انتها می‌رسه ولی معماری کامپیوتر می‌تونه کامپیوترهای بهتری طراحی کند

الان مسئله اصلی در طراحی مصرف انرژی است و معماری کامپیوتر علاوه بر performance باید به مصرف انرژی هم توجه کند.
امروزه دنیای نرم‌افزار برای اینکه بتواند از آن تکنولوژی که هست بهترین استفاده را بکند باید خودش را با تحولاتی که
در دنیای Computer Architecture رخ می‌دهد وفق دهد تحولاتی که ایجاد شد اینکه معماری کامپیوتر از single core به
multi core رفت پس software باید parallel programming کند و قدرت پردازشی که معماری کامپیوتر برایش ایجاد
کرده به نحو احسن استفاده کند. واحد پردازشی به نام GPU طراحی کردند که ظاهر نرم‌افزار دیگه نمی‌تونه به سبک سنتی قبلی
کار انجام دهد و باید زبان برنامه‌نویسی مطابق آن را انجام دهد

اسلاید ۱۲:

می‌خواهیم سیستم‌های کامپیوتری که الان داریم چیه هستند؟

۱) موبایل‌ها و تبلت‌ها: در سال بیش از ۱ میلیارد به فروش می‌رسند و پرمصرف‌ترین بازار کامپیوترها هستند. بازار مصرف‌شون
دست پردازنده‌های مبتنی بر arm هست که این arm‌ها در دل یک chip بزرگ هستند در کنار arm یکسری پردازنده
دیگه هستند مثل پردازنده ترافیک، پردازنده ویدیو، GPS و ...

۲) کامپیوترهای بزرگ (سرورها، Data Centers): هزاران واحد پردازشی دارند، حجم انبوهی از رم‌ها را دارند.
بازار مصرف‌شون عمدتاً دست پردازنده‌های x86 هست. در این سیستم‌ها ایلیاس‌های خاص خودشون را دارند و
پردازنده‌های عام منظوره دارند در کنار آن‌ها یکسری پردازنده خاص منظوره مثل GPU و FPGA هم وجود دارند که
یکسری کارهای خاصی را انجام می‌دهند

Scanned by CamScanner

Year: ..... Month: ..... Day: .....

۳) سیستم های مخفف: هر Device غیر از موبایل، کامپیوتر و لپ تاپ و تبلت و server و Data center

که کامپیوتر داخل آن باشد هستند مثل ماشین لباسشویی، یخچال، اتومبیل، ساعت و .... (هر آنچه که با برق کار می کند)

اسلاید ۱۳:

تاریخچه ای از رایانه ها کامپیوتر را می گوید

اسلاید ۱۴ و ۱۵:

کامپیوترها دو دسته هستند: آنالوگ و دیجیتال

اولین کامپیوتر شکلی است که در این اسلاید است (به صورت تکاملی آنالوگ) و برای محاسبات نجومی و نظامی استفاده می شده

یا به عبارت دیگر شریک هواپیمایی که برای ارزیابی بال ها استفاده می کردند

در هواپیمایی که آنالوگ دقت محاسباتی خوبی ندارد

اسلاید ۱۶:

کامپیوترهای دیجیتال که با عدد و رقم سر و کار دارند نمایش می دهند که اعداد گسسته هستند و قدرت محاسباتی قطعی به ما می دهند

هر ماشینی که قابل برنامه نویسی باشد می تواند انجام دهد و برنامه نویسی کند

« جلسه دوم »

ویدیو اول

اسلاید ۴:

مروری بر مطالب جلسه قبل

تفاوت زبان سطح بالا و سطح پایین اینکه زبان سطح بالا وابسته به معماری و IZ و ... برداشته ظاهری نیست

در حالیکه وقتی در اسمبلی کد می نویسیم برای یک معماری خاص می نویسیم و تاثیرش اینکه وقتی کل کاری در دایلکشن

در هم تنیده باشد با معماری به راحتی می توانیم معماری را هر روز عوض کنیم و در سمتون بازنشسته که هر روز update می کنیم و

نرم افزار خودشو با خاص و من بده ولی وقتی در زبان سطح بالا برنامه می نویسیم اون وقت اهمیتی نداره معماری چی هست

و سازنده نرم افزار هم هر تغییراتی که بخواد می تونه انجام بده بدون اینکه نگران باشه نرم افزاری که با این کار می کنه مستقیم بر استون

پیش بیاد و OS ها هم دیگه وابسته به معماری نیستند

دو نوع معماری داریم RISC و CISC ← توانایی زیادی به برنامه نویس می داد ولی پیچیده بود

پیچیده نیستند و IZ های ساده داشتند و برای کارهای پیچیده باید آن را تبدیل کنیم و ساده کنیم

Scanned by CamScanner

تکنولوژی و معماری بهتر باعث شد که ما کدنویس رمان کامپیوترهای بهتر، از آن تر و سریع تر داشته باشیم که بتوانند کارایی بهتری هم به ما بدهد و زبان های برنامه نویسی هم بسیار تقویت شدند

Year: Month: Day:

اسلاید ۶:

- بیشترین تلاش در معماری در دهه رمان اوایل در دهه ۸۰ و ۹۰ به سمت این بود که در سطح instruction موازات بیشتری تامین کنیم

موازات در سطح دستورات (ILP) یعنی پردازنده را تا حد کنیم که در هر لحظه از زمان در پیش بینی یک instruction باشد (معماری هافیومن به این صورت نبود که به صورت موازی انجام دهد) یعنی به صورت pipline انجام شود

ILP تقریباً از ۵ سال پیش اشباع شد یعنی بیشتر از یک اندازه نمی توانستیم مجبور به موازی سازی کنیم

- وقتی ILP به انتها رسید و نتوانستیم موازات در سطح دیتا در این مدل سعی می کنیم دیتا ها را هم به صورت موازی انجام دهند (DLP) همراه با اینکه دستورات موازی انجام می شوند

بالاتر از این موازات در سطح thead یعنی یک thread را موازی می کنیم با چند thread دیگر (TLP)

بالاتر از این RLP هست یعنی کامپیوتر در هر زمان بتواند تا تا Reague را هندل کند (در سرورها بیشتر کاربرد دارد)

- وقتی از ILP به سمت DLP، TLP و RLP می آییم با سطح برنامه نویسی را هم تغییر می دهیم در ILP تعاریفی که با شبیه داره اینکه مقدار کامپیوتر وقتی داره ILP را اجرا می کند زیر اسم اولین بی خطر و برای فرقی نمی کند داره instruction ها موازی انجام میشن یا سریال اما وقتی پردازنده ها قابلیت DLP، TLP و RLP داره به شرطی قابل استفاده است که برنامه نویسی هم تغییر کند و برنامه ها را به شرط اینکه این قابلیت ها را دارد Develop کنیم

اسلاید ۷:

- دسته بندی کامپیوترها از نظر Flynn:

۱) کامپیوترها یا SISD هستند که در این حالت یک مسیر از instruction ها پشت سر هم میرن و در هر لحظه یک instruction داره کارش را می انجام و یک دیتا را داره تولید می کند

۲) SIMD: یک مسیر instruction میره و n تا مسیر Data دارن میرن و هر instruction با n تا دیتا که داره کار این میشه یا پردازنده برداری هستند

- multimedia extension هستند (در اینتل استفاده می شد)

- GPU هستند

این نوع کامپیوترها هایی پردازنده های برداری هستند که وقتی می خوان عملی انجام بدن برخلاف عملیات عملی که در پردازنده انجام می شد در GPU یک ماتریس a را با ماتریس b جمع می کنیم و می ریزیم در ماتریس c که این توسط یک instruction انجام می شد در حالیکه اگر همین کار را روی پردازنده های اینتل بخواهیم انجام بدیم باید در یک loop دایره بزنیم

Scanned by CamScanner

ماتریس‌ها را با هم جمع کنیم و این instruction جمع را n بار اجرا کنیم در حالی که در Cpu یک بار انجام می‌دهیم

Year: ..... Month: ..... Day: .....

۳) MISD: در این معماری instruction In جریان می‌یابد جلو و یک دیتا تولید می‌شود که وجود خارجی ندارد این

۴) MIMD: در این حالت در هر لحظه instruction In داریم هر کدوم دیتای خودشون را دارن

این حالت دو حالت دارد

Tighly-Coupled MIMD: ارتباطی که بین واحدهایی که دارن MIMD را اجرا می‌کنن ارتباط مستحکم و نزدیکی است که حالتی است که در multi core ها داریم

Loosely-Coupled MIMD: ارتباط سست و اندکی وجود دارد که واحدهای پردازشی multi computer هستند (هر بخش از پردازنده را یک کامپیوتر انجام می‌ده)

اسلاید ۸:

کامپیوتر در دنیای کامپیوتر از منظر ساختاری همه پایین یک هسته پردازشی داریم (L1 cache) در سطح chip ساخته می‌شه که در اون pipeline، hazard، superscaler و .... را داریم. هر core یک cache و Isa

مخصوص به خودش دارد که این core یک L1 cache را (کش سطح اول) را دارد

در level بالاتر کش سطح ۲ را داریم و در بعد از آن حافظه DRAM یا main memory را داریم

هر کدوم از این سطوح مسائل و مشکلات متفاوتی دارند

در سطح بالاتر حافظه جانبی مثل Disk و SSD و کلا انواع و اقسام حافظه‌ها را داریم

اسلاید ۹:

معماری کامپیوتر اگر در گذشته وجود داشته باشد کامپیوتر وجود داشت این بود که بتوانیم Isa طراحی کنیم یعنی طراحی Isa برای کامپیوتر را معادل معماری کامپیوتر می‌دانستند

وقتی Isa طراحی می‌کنیم مسائلی که داریم اینکه چند تا رجیستر داشته باشیم، چند بیتی باشن، آدرس دهی به حافظه چه جوری باشد.....

اینها مسائلی است که حل شده و وظایفی که ما به عنوان معماری کامپیوتر باید انجام بدهیم اینکه، نیازمندی‌های یک کامپیوتر در اپلیکیشن خاص، چقدر قدرت پردازش باید داشته باشیم، cache را چگونه بدهیم، چه عملیاتی را سیستم‌ها پشتیبانی کنیم، فرکانس و ورودی‌ها را تنظیم کنیم و در طراحی اینجوری نباشه که هر ماه دستورات بیشتری داشته باشیم، فرکانس را ببریم بالا تا کارایی بیشتری داشته باشیم

سعی که در مقابل کارایی که داریم باید جوابمون به هزینه، مصرف انرژی و ... باشد (علاوه بر اینکه شامل Isa، hardware

micro Architecture می‌شه باید این موارد را هم در نظر بگیریم

Scanned by CamScanner

اسلاید 10

Year:..... Month:..... Day:.....

از نظر تکنولوژی طبق قانون مور: density ۲۵٪ به بهبود می‌آید ،

Die size (اول) تراشه‌ای که می‌خواهیم ترانزیستورها را در آن جا بدهیم) ۱۰ تا ۲۰ درصد سالانه بیشتر می‌شه

مجموع ۴۰ تا ۵۰٪ integration ما بیشتر می‌شه یعنی مدارها مجتمع‌تر می‌شن و یک IC ترانزیستور بیشتری

داشته که این بهبودی باعث می‌شه کارایی بیشتری داشته باشیم

- در کنار واحد بازیابی حافظه‌ها هم که باید سرعتش را با پردازنده تطبیق بده DRAM ها که تکنولوژی قالب حافظه‌ها

هستند (تنها تکنولوژی حافظه‌ها) از نظر ظرفیت ۲۵ تا ۴۰ درصد زیاد می‌شن

در سال ۲۰۱۴ DRAM ، 8Gb بود که ۵ سال بعد DRAM که توسعه بسازن 16Gb است که 32GB را ممکنه تو بسازن

حافظه دیگه Flash است که ۵۰ ، ۶۰ برابر افزایش داشته که از نظر قیمت Flash ، 8 یا ۱۰ برابر ارزان‌تر از DRAM است

- حافظه دیگه برای استفاده hard disk یا magnetic disk است که سرعت رشدشون خیلی کم شده و ۵ برابر شده

از نظر قیمت ۱۰ برابر از Flash ارزون‌تره از DRAM هم ۱۰۰ برابر ارزان‌تر است ، سرعتش نسبت به SSD تنها

سه‌تش است

اسلاید 11

پهنای باند و تأخیر

پهنای باند حافظه

پهنای باند تعداد کارهایی که در واحد زمان انجام شده مثلاً در واحد زمان چقدر داده به پردازنده می‌تونیم بدیم ، پردازنده چه حجمی

از اطلاعات را از حافظه می‌تونه بگیره ، پردازنده بازمن اینکه تعداد دلخواه ورودی بگیره چقدر رمز می‌تونه بده

پهنای باند پردازنده — پهنای باند

پهنای باند پردازنده‌ها ۳۰ تا ۴۰ هزار برابر شده اما پهنای باند memory و DISK ، ۳۰۰ تا ۱۲۰۰ برابر شده

تأخیر: فاصله زمانی بین شروع تا پایان یک کار ، تأخیر در پردازنده ۵۰ تا ۹۰ برابر شده در DISK و حافظه

۶ تا ۸ برابر شده

ارتباط پهنای باند و تأخیر: سامانه‌ای پردازنده‌ی تأخیرش n برابر بهبود پیدا کند پهنای باند هم n برابر بهبود می‌یابد ، اگر

پهنای باند n برابر شده باشه می‌شه تأخیر $\frac{1}{n}$ شده

Scanned by CamScanner

Year: ..... Month: ..... Day: .....

اسلاید ۱۲:

این شکل می‌گه حافظه، دیسک، پردازنده و شبکه با چه سرعتی دارند رشد پیدا می‌کنند. محور x تأخیر و محور y پهنای باند است

اون خط قرمز صاف می‌گه اگر تأخیر ۱۰ برابر بهتر شده باشه پهنای باند هم ۱۰ برابر نشون می‌ده که اون منحنی بالای این خط نشون می‌ده تأخیر هر چقدر زیاد شده پهنای باند چقدر بهبود یافته

مثلاً در شبکه تأخیر ۱۰ برابر بهتر شده ولی پهنای باند ۵۰ برابر بهتر شده یعنی حجم دیتاهایی که می‌تونیم بفرستیم ۱۰۰ برابر بهتر شده

نکته: این نمودار بهبود تأخیر به تحول نسبت باعث بهبود پهنای باند شده (هر دو n برابر شدن) ولی عکس این موضوع درست نیست بهبود پهنای باند n برابر شده تأخیر ما $\frac{1}{n}$ ام نشده

نکته: اگر همه از یک نقطه شروع کنن ۳۰، ۴۰ سال بعد پهنای باند cpu ۳۰، ۴۰ هزار برابر شده، پهنای باند شبکه هم همین طور ولی پهنای باند memory بین ۱۰۰۰ تا ۱۰۰۰۰ است و پهنای باند DISK بین ۱۰۰ تا ۱۰۰۰ است (نسبت پیشرفت یکسان نبوده و حافظه از شبکه و cpu عقب مونده)

اسلاید ۱۳:

* تکنولوژی این اجازه رو می‌ده که Feature size (ضخامت ترانزیستور) را کاهش دهیم و به نصف کاهش Feature size آخیر کاهش پیدا می‌کند

- در سال ۱۹۷۱ اینتل اولین پردازنده در IC را ساخت با تکنولوژی 10 micron و در سال ۲۰۱۷ با تکنولوژی ۱۱ نانو می‌سازه یعنی ۱۰۰۰ برابر Feature size کاهش پیدا کرده.

- از نظر کارایی سرعت کلاک از نظر خطی با Feature size تغییر می‌کنه یعنی Feature size نصف بشه سرعت هم نصف می‌شه

- از نظر integration (انبوه سازی و جادادن ترانزیستورها در مساحت مشخص) توان ۲ برابر افزایش می‌یابد نسبت به Feature size

اسلاید ۱۴:

در طراحی پردازنده علاوه بر کارایی باید حواسمون به انرژی مصرفی و توان هم باشد که باعث TDP پیش می‌آید.

TDP: وقتی یک پردازنده را طراحی می‌کنیم پردازنده دارد انرژی مصرف می‌کند و انرژی به گرما تبدیل می‌شه

دمای chip بالا می‌ره که در طراحی باید حواسمون باشه سرعت انتقال گرما به بیرون با گرماهای داخل برابر باشه

TDP می‌گه ما باید ماکزیمم حجم گرمایی که توسط یک chip می‌تونه تولید شود را بدست آوریم وقتی می‌خواهیم سیستم خنک کننده طراحی کنیم باید حواسمون به این مقدار باشد

TDP به طور خلاصه یعنی اندازه‌گیری کنیم ماکزیمم توان یک تراشه می‌تواند مصرف کند را در بیاریم و برمبنای اون

Scanned by CamScanner

طراحی کنیم

Year: ..... Month: ..... Day: .....

این ماکزیمم توان را از کجا بدست آوریم؟

یک chip زمانی ماکزیمم توان را دارد که تمام سمت هاش در حال کار باشن (در هر کلاک تمام ترانزیستورها را خاموش روشن کنیم)

TDP میشه احتیاج نیست این مقدار را در نظر بگیریم چون هیچ وقت این اتفاق نمی افته بلکه باید میانگین مصرف انرژی را به عنوان مقدار توان مصرفی بگیریم ولی میانگین عدد درستی نیست چون خیلی وقت ها بیشتر میانگین پردازنده مصرف می کند. TDP میشه مدار یک peak power داره که نه اون حالت بدبینانه است نه اون حالت میانگین چیزی بین این هاست حالتی است که مدار بیشترین حجم کاری را که می تواند انجام بدهد است. اون حالت بد که ۵ برابر مقدار میانگین است را مقدار peak power قرار بدیم را اجازه نمیدیم از این مقدار بیشتر بشود

- علاوه بر مقدار TDP که باید حواسمون باشه ما می توانیم به صورت داینامیک ترانزیستور و chip را مصرف انرژیشون را کنترل کنیم با تکنیک هایی که به کار می بریم مثلاً می توانیم یک پردازنده که کلاکش یک عدد مشخصی است می توانیم برحسب ضرورت فرکانس کلاک را بیاریم پایین که در این صورت مقدار پردازش کاهش می یابد در عوض انرژی کمتری مصرف می کند در این صورت گرما کمتری تولید می کند بنابراین ترانزیستور اگر سینه داره داغ می شه فرکانسشو می تونه کاهش بده.

- اگر بخواهیم با کلاک بازی کنیم، آیا می تونیم بگیم سیستم انرژی کمتری مصرف می کند؟

در کامپیوتر می گیم اون کامپیوتری که توان کمتری نسبت به دیگری مصرف می کند پردازنده بهتری نیست بلکه اون بهتر است که به ازای کاری که انجام میدن انرژی کمتری مصرف کند (اونی بهتر است که Energy per task بهتری دارد)

اسلاید ها:

انرژی توان به دو قسمت تقسیم می شوند:

انرژی داینامیک ← متناسب است با تعداد switching ها در مدار. هر چقدر بیشتر ← 0 و 1 ← شود انرژی بیشتر مصرف می شه

فرمول: $\frac{1}{2} C V^2$ → ولتاژ

خازن جایی که سوئیچ می شد

Scanned by CamScanner

توان دینامیک: انرژی در واحد زمان

فرمول: $P = \frac{1}{2} C V^2 F$ → فرکانس (تعداد سوئیچ ها در واحد زمان)

Year: ..... Month: ..... Day: .....

طبق فرمولی که برای توان و انرژی داریم اگر ما clock rate را پایین بیاریم، فرکانس را کم کنیم توان ما کمتر می شه اما کاهش فرکانس هیچ تأثیری در مصرف انرژی ندارد

اسلاید ۱۶:

پردازنده اینتل ۸۰۳۸۶ توان مصرفش در حد ۲ وات بوده

پردازنده Core i7 اینتل که 3,3 GHz فرکانسش هست مصرف توان ۱۳۰ وات است که این ۱۳۰ وات گرمایی تولید می کند در ابعاد $1,5 \times 1,5$ cm و ما باید بتوانیم این گرما را سریع منتقل کنیم (هر چی مساحت کمتر باشه گرما بیشتر در اونجا بالا میره)

اسلاید ۱۷:

تکنیک برای کاهش مصرف توان و دما:

Core را خاموش کنیم تا کمی سرد تر شود

از تکنیک DVFS استفاده کنیم که هسته سرعت را کاهش بده تا مجبور نباشی در اثر گرمای زیاد متوقف بشی

در این شکل می شه ما فرکانس را می توانیم براساس حجم کاری که داریم تنظیم کنیم تا گرمای زیاد تولید نشه اگر کار کمی داریم

این کاهش توان برای دیسک پردازنده بود برای کاهش مصرف انرژی در DRAM و Disk می توانیم در زمان هایی که کاری ندارن آنها را خاموش کنیم یا در حالت low power قرار بدیم

تکنیک دیگه اینه یک پردازنده که فرکانس کاری مشخصی داره این قابلیت را دارد که فرکانسش یک مقداری از آن بالاتر بره کارکرد به اصطلاح over clock کنیم که کار را سریع انجام دهد و بعد خاموش کنیم که خیلی گرم نشه

اسلاید ۱۸:

روشن بودن سیستم بدون فعالیت هم انرژی مصرف می کند که به عنوان power static می شناسیم که می شه انرژی که در واحد زمان سیستم مصرف می کند فارغ از این که کاری انجام بده یا نده چون ترانزیستورها یک جریان نشتی دارند

این power static وقتی ترانزیستورها کوچک می شن نشتی مهم تر می شه و نسبت به انرژی کل بالا می ره و تا حدی ۵۰٪ مصرف انرژی در سیستم به خاطر همین کاری است که انجام می دیم که تابع ولتاژ جریان static است اگر ما ولتاژ را پایین بیاریم جریان کم می شه

بسیار کاملا خوبی داره با مساحت و تعداد ترانزیستورها و راه کاهش آن اینه از تکنیک power gating استفاده کنیم یعنی

Scanned by CamScanner

هر قسمتی از مدار که کارشی نداریم power آن را قطع کنیم که برای این کار باید power را and کنیم با یک سیگنال Enable
که وقتی کار داریم Enable را یک کنیم و وقتی کار نداریم صفر کنیم

Year:..... Month:..... Day:.....

ویدئو دوم

اسلاید ۴:

- معیار performance:

۱) زمان پاسخ. معادل تأخیر

۲) نرخ بهره معادل پهنای باند

- وقتی می‌خواهیم کارایی را بهتر قرار دهیم کارایی یک سیستم را با سیستم دیگر می‌سنجیم، سرعت یک سیستم را با دیگری می‌سنجیم

$$\text{speedup} = \frac{\text{زمان اجرای کار در } y}{\text{زمان اجرای کار در } x}$$

بهبود یافته x است

- زمان اجرا: می‌تواند زمان سپری شده برای انجام کار باشد (wall clock time) مثل این روش اینکه سیستم در شرایط مختلف کارها را در زمان‌های متفاوتی که انجام شده (ممکن سیستم سرش شلوغ باشه و انجام کارها بیشتر وقت احتیاج داشته باشد)

معیار بهتر cpu time است یعنی بازه‌هایی که cpu فرض کار ما شده است را فقط در نظر بگیریم

- زمان اجرای سیستمی که می‌خواهیم اندازه‌گیری کنیم: در واقع زمان اجرای چه چیزی را می‌خواهیم اندازه بگیریم، چه چیزی را باید در نظر بگیریم

✓ بنچمارک برنامه‌هایی که روی هر سیستمی وجود دارد مثل ضرب ماتریس را بهش بدیم

✓ می‌توانیم برنامه‌های ساده مثل sort کردن یک برنامه را در نظر بگیریم

✓ می‌توانیم برنامه‌های خودساخته را بهش بدیم

✓ راه اصولی این است که از سته‌های محک benchmark ها استفاده کنیم (اپلیکیشن‌های استاندارد که سیستم را در شرایط مختلف چک می‌کنند و اندازه‌گیری می‌کنند)

اسلاید ۵:

cpu time: تایمی که سیستم فرض برنامه ما کرده

$$\text{cpu time} = \frac{\text{کلاک‌هایی که صرف برنامه ما کرده}}{\text{نرخ کلاک}}$$

عکس فرکانس

$$\text{cpu time (فرکانس)} = \text{کلاک‌هایی که صرف برنامه ما کرده} \times \text{زمان هر کلاک}$$

Scanned by CamScanner

CPI = تعداد کلاک هایی که صرف برنامه کرده / تعداد instruction های برنامه = CPU clock cycles for a program / instruction count ← می‌گه به ازای هر instruction چند کلاک

Year: Month: Day:

عدد جمع شده این پردازنده

CPI & CPU time محاسبه ⇒ CPU time = تعداد دستورات × تعداد کلاک به ازای هر دستور × زمان هر کلاک

اسلاید ۶:

در مورد CPI نکته‌ای وجود داره اینکه برنامه دستورات مختلف داره و دستورات مختلف پیچیدگی‌های معماری متفاوتی دارند و تعداد کلاکی که خرج هر دستور می‌شه متفاوت است

(CPU clock cycles) تعداد کل کلاک هایی که خرج کردیم = $\sum_{i=1}^{n} IC_i \times CPI_i$ به ازای هر دستور یک CPI داریم

هر دستور یک فراوانی داره در برنامه

$$CPU\ time = \left(\sum_{i=1}^{n} IC_i \times CPI_i\right) \times clock\ cycle\ time$$

اسلاید ۷:

تکنیک‌های برای طراحی با کارایی بهتر:

قانون ۱) تا می‌توانیم موازات داشته باشیم یعنی به جای یک پردازنده چند پردازنده داشته باشیم، به جای یک دیسک n دیسک داشته باشیم، در حافظه به جای یک بانک حافظه n تا بانک حافظه بذاریم، از pipling استفاده کنیم در ALU ها ضرب شده جمع شده همزمان چند تا تبدیل کار کنند

قانون ۲) از اصل مجاورت استفاده کنیم: این اصل میگه تا می‌تونیم Data ها و instruction ها را باز استفاده کنیم چون این اصل میگه داده ای که الان داریم استفاده می‌کنیم به احتمال زیاد در آینده نزدیک بلافاصله و کاری هاش کار داریم

قانون ۳) سعی کنیم تمرکز و هزینه‌ها یون را جاهایی خرج کنیم که پرمصرف هستند. فرض کنیم در پردازنده یک فضای خالی وجود داره می‌تونیم یک ضرب کننده Floating point بذاریم یا جمع کننده integer بذاریم

این قانون میگه اول فضا را صرف کسی کن که استفاده بیشتری داره به جمع کارایی خیلی بیشتری از ضرب داره پس این فضای خالی را به جمع بده

Scanned by CamScanner

قانون ۴) قانون آمدال که می‌گه اگر می‌خواهیم از سیستم کارایی بگیریم

Year:..... Month:..... Day:.....

$$\text{Execution time}_{new} = \text{Execution time}_{old} \times \left((1 - \text{Fraction}_{enhanced}) + \frac{\text{Fraction}_{enhanced}}{\text{speedup}_{enhanced}} \right)$$

$$\text{speedup}_{overall} = \frac{1}{(1 - \text{Fraction}_{enhanced}) + \frac{\text{Fraction}_{enhanced}}{\text{speedup}_{enhanced}}}$$

اسلاید ۸:

جمع‌بندی این فصل

« ویدئو جلسه ۳ »

ویدئو اول (Lec3.a)

(حافظه، ساختار و سلسله مراتب حافظه)

اسلاید ۴:

- انتظاری که یک برنامه‌نویس از حافظه داره اینه یک فضای بی‌نهایت داشته باشه برای ذخیره هر آنچه که داره و تأخیرش هم کم باشه و به سرعت بتونه پردازنده را تغذیه کنه. برنامه‌نویس می‌خواد مدام نگران محدود حافظه باشه
- حافظه‌ای بخواد تأخیرش کم باشه و سرعتش بالا باشه حافظه گران‌تر است و اینقدر که در رابطه با حافظه‌ها وجود داره اینکه هر چقدر بخوان سریع‌تر بشن گران‌تر هم می‌شن که می‌شه هزینه زیادی کرد
- ما به عنوان طراح حافظه سلسله مراتب حافظه یا memory hierarchy را معرفی می‌کنیم به صورتی که کل فضای آدرس‌دهی حافظه، آن حافظه‌ای که برنامه‌نویس می‌بینه خیلی بزرگ است ولی نه اینقدر تند که سرعت کم را انتخاب بکنیم ولی حافظه زیاد را انتخاب می‌کنیم مسئله‌ای که الان داریم سرعت خیلی زیاد با طراحی سلسله مراتب حافظه سطوح مختلف حافظه را از حافظه‌ای که پردازنده می‌بیند خیلی سریع حافظه‌هایی در level های مختلف می‌چینیم تا برای برنامه‌نویس دسترسی سریعی باشد
- پس ما چنین معماری را می‌بینیم بعد از ایجاد آن حافظه‌ی خیلی بزرگ، بعد از آن یک حافظه کوچکتر و سریعتری داریم، بعد از آن باز هم یک حافظه کوچکتر و سریع‌تری داریم تا این جوری پیش می‌رویم تا به رجیستر پردازنده می‌رسیم که در کنار یک حافظه خیلی سریع، سریع‌ترین تکنولوژی حافظه را می‌گذاریم که خیلی کوچک است، کوچک بودن آن مسئله‌ای که ایجاد می‌کنه اینه که اون حجم انبوه اطلاعات را نمی‌تواند نگهداری کند پس کاری که باید بکنیم اینه که با بهره‌گیری از مکانیزم‌هایی که به عنوان یک واحد Access حافظه وجود دارد این تضمین را برای برنامه‌نویس و طراح ایجاد می‌کنیم که تقریباً تمام Request هایی که به حافظه داره (بدون هر آنچه که در درون حافظه وجود دارد) می‌تونیم توسط آن

Scanned by CamScanner

حافظه کوچک برایش فراهم کنیم این توهم را برای برنامه نویس ایجاد می کند که یک حافظه بزرگ با سرعت بالا داره. در واقع ما اطلاعات را در حافظه بزرگ نگهداری می کنیم هر زمان به آن احتیاج شد به حافظه کوچکتر منتقل می کنیم

Year:..... Month:..... Day:.....

اسلاید ۵:

ساختار حافظه (سلسله مراتب حافظه) که در دنیای امروزی وجود دارد:

حالت A: ما یک CPU داریم که در دل آن یکسری رجیستر وجود دارد که رجیسترها سریع ترین و کوچکترین حافظه ای موجود هستند. بعد از آن یک مجموعه حافظه کش داریم که از دنیای پردازنده که خارج میشویم سریع ترین و کوچکترین حافظه است. بعد از آن مجموعه ی دوم کش را داریم که بزرگ تر است و البته کندتر. مجموعه اول در حد ۶۴ کیلوبایت است و Access time=1ns دومی ۲۵۶KB و حدود ۵۱۲KB Access time=۵ تا 10ns بعد می شه min memory 10 در حد 1 تا 2G است و Access time=۵ تا 100ns بعد از آن حافظه بزرگ قرار دارد که از جنس Flash memory است و حجم آن ۴ تا ۶۴ گیگابایت و Access time در حد میکروثانیه است. این ساختار برای موبایل ها بیشتر مورد استفاده است

حالت B) کامپیوترهای دسکتاپ و لپ تاپ سلسله مراتبشون تفاوت داره اینکه یک level کش 3 تا بین L2 و memory قرار می گیره که حجمش در حد چند MB است و Access time هم دو سه برابر L2 است و حافظه بزرگ هم می تونه از جنس Flash (SSD) باشه یا از جنس HDD یا هارد دیسک هم باشه

حالت ۳) در سرورها استفاده می شن، مثل دسکتاپ هستن تقریباً فقط حافظه سطح آخر یکی از SSD یا هارد دیسک می تونه باشه (مقدار حافظه و Access time هر کدوم پایینش هست)

اسلاید ۶:

این نمودار نشون می ده پردازنده و حافظه سرعت رشدشون به چه صورت است. پردازنده از سال ۱۹۸۰ تا ۲۰۰۵، ۱۰ هزار برابر شده ولی memory دو برابر هم نشده. در این سال ها performance حافظه (چون یک دیتا) در memory دو برابر کم شده در حالیکه پردازنده بیشتر هم شده چون از یک فرکانسی نمی تونه بالاتر بره. این نمودار نشون می ده که اهمیت حافظه بالاست ولی از پردازنده عقب می افته. کسی که باید این عقب افتادگی را دنبال کند تکنولوژی نیست چون این نمودار نشون میده تکنولوژی نمی تونه کار حافظه را درست کند که این کمبود حافظه را باید معماری کامپیوتر پر بکند و مجبور بنشه

اسلاید ۷:

- ما نمودار 1-core و پردازنده را دیدیم ولی امروزه پردازنده های به سمت multi-core رفتند تا بتونند قدرت پردازش را با core های بیشتر بالاتر ببرند مثلاً با 4-core، 4 برابر کند که این باعث می شه memory

Scanned by CamScanner

از این که هست عقب تر هم نیست

Year: Month: Day:

- یک مثال برای این مورد؟

یک پردازنده core i7 اینتل را در نظر بگیریم، این پردازنده در هر ثانیه به ازای هر core یک Request می‌تونه بده، هر core فرکانس ۲/۳ داشته باشه این پردازنده در هر ثانیه می‌تونه یک Request بده به یک instruction اجرا شده یک instruction دو operand ورودی داره بنابراین پردازنده در واحد زمان می‌تواند ۲۵/۶ میلیارد درخواست دیتای ۶۴KB داشته باشد + ۱۲/۸ میلیارد instruction ۱۱۸ bit ریکوئست بده

مجموعاً همه اطلاعاتی که می‌تونه داشته باشه ۴۹/۶GB/s

حافظه فقط ۸ درصد این عدد را می‌تونه تأمین کند و جواب بده

این ما را ملزم می‌کند که برویم به سمت کش‌هایی که مثل کش‌های سنتی نباشند بلکه به صورت pipe line عمل کنند

یعنی در هر لحظه از زمان یک Request را به دنبال دیتا می‌برده یکی را دنبال tag می‌برده، برای یک Request دیگه که miss شده دیتا بیاره، از طرف دیگه باید multi port باشه یعنی ما حافظه‌ای که به صورت مشترک داریم اینه که یک port آدرس می‌ده یک port دیتا می‌ده ولی حافظه multi port حافظه‌ای است که بیشتر از یک port آدرس و دیتا دارد و در هر لحظه از زمان می‌توانیم چند تا ریکوئست بدهیم به اضافه این که در ساختار پردازنده‌های multi core دو level کش داریم به ازای هر core که داریم پس یک L1 cache تا ۴ core و L2 cache تا ۴ داریم یک level 3 کش هم داریم که مشترک است با این ۴ تا core

اسلاید ۸۸

وقتی همچین پردازنده‌هایی می‌خوای ساختاری را می‌چینیم روی پردازنده یعنی سه level کش که کش L3 در حد بالای L3 یا L1 قرار گیاه قلعه است و این قلعه روی chip ما را می‌بره این می‌چاپش از نصف chip ما رو می‌گیره و از نظر area و توان مقدار زیادی صرف حافظه‌هایی می‌شه که روی chip کنار پردازنده قرار می‌گیره چرا اینقدر هزینه می‌کنیم برایش؟ چون می‌خوریم اینقدر هزینه کنیم تا راه دور و درازی را تا main memory نزنیم، فاصله برای این مهم است که هر چقدر بیشتر باشه تأخیر و زمان رفت و برگشت زیاد می‌شه

اسلاید ۹۰:

برای اینکه بتوانیم ساختار حافظه را optimise کنیم باید بدانیم معیارهامون از حافظه چی هست؟

- معیارهامون از performance چی هست، ما وقتی یک cache و حافظه را طراحی می‌کنیم خواسته‌مون latency باشه، مورد دیگه اینکه پهنای باند چقدر باشد از نگاه کسی که بیرون نشسته و input و output یک

Scanned by CamScanner

حافظه را می‌خواهیم پهنای باند مهم است نه latency

Year: Month: Day:

* معیاری که از نظر performance ، access time است یعنی فاصله زمانی بین زمان درخواست خواندن تا زمانی که داده در دست ما می‌رسد (هر درخواست چقدر طول می‌کشد)

یک مورد دیگه cycle time است که می‌گه مینیمم زمان دو درخواست که پشت سر هم به آن یعنی یک درخواست داده درخواست بعدی را چه زمانی می‌توانی بدی

- حافظه‌ای که سرعتش زیاد باشد و latency کم باشد SRAM است که برای کش استفاده می‌شه

- برای main memory از DRAM استفاده می‌کنیم و این DRAM برای اینکه پهنای باندش بالا برود در بانک‌های حافظه می‌چینیم یعنی به جای این که یک حافظه بزرگ یک دست داشته باشیم حافظه را می‌شکونیم به چندین بانک حافظه که بانک حافظه‌ها می‌توانند مستقل از بقیه access بشن و وقتی به یک بانک دسترسی پیدا می‌کنیم به بانک‌های بعدی هم می‌توانیم به موازات این درخواست بدهیم

اسلاید ۱۰:

در تکنولوژی SRAM هر بیت ۶ تا ترانزیستور لازم داره و کافی ترانزیستورهاروش باشن که داده را در خودش نگه داره بنابراین برای نگهداری داده انرژی و توان زیادی مصرف می‌کنه

در DRAM قابلیتی که با SRAM داره اینه که هر بار که ما ازش می‌خوانیم دیتاش از بین می‌ره که دوباره باید در آن بنویسیم که در این صورت باید انرژی و توان زیادی مصرف بشه به ازای write که انجام می‌شه + زمانی که صرف نوشتن می‌شه

پس DRAM از SRAM کندتر است از زمان مصرف انرژی هم بیشتر است ولی بستر و جایی که نقش پررنگ‌تره در DRAM اگر کاری هم باهاش نداشته باشیم به صورت پریودیک باید Refresh کنیم و دوباره بنویسیم چون خود به خود دیتا از بین می‌ره ، ۵٪ از زمان‌هایی که DRAM استفاده می‌شه به دلیل همین Refresh است

که هزینه‌ای که DRAM داره اینه که یک ترانزیستور برای هر بیت می‌خواد (برای نگهداری هر مفهوم یک خط یک ترانزیستور می‌خواد)

پس DRAM از SRAM چگال‌تر است در واحد سطح حجم بیشتری از DRAM می‌توانیم داشته باشیم تا SRAM

اسلاید ۱۱:

اعمال می‌شه:

- نگاشت حافظه باید به صورت خطی با سرعت پردازنده بره بالا ولی واقعیت اینه که این دو تا خیلی با هم match نشدن یعنی اندازه که این سرعت و قدرت پردازنده رفته بالا به قول نسبت نگاشت بالا نرفته

Scanned by CamScanner

- برای این که عقب افتادگی حافظه را جبران کنیم از چند تکنیک استفاده می‌کنیم:

Year:..... Month:..... Day:.....

۱) در هر ردیف حافظه RAM یک بلاک کش را به پردازنده می‌دهد که Request داده می‌دهیم و

Access های بعدی پردازنده اگر جوری باشد که نزدیک همون بلاک قبلی باشد در همون ردیفی هستند

که الان خوندیم ازش

۲) از SDRAM استفاده می‌کنیم (DRAM که سنکرون است) یعنی DRAM که کلاک دارد و سرعتش بیشتر است

خودمان می‌توانیم از دو تا تکنیک Access به صورت Burstmode و Access به صورت Critical word

هم استفاده کنیم

- کاردینالیته که می‌گفتن RAM interface را عرض می‌کنن یعنی اگر RAM مثلا ۳۲ بیت بوده الان ۶۴ بیت کردن

- یک دیگه تکنولوژی DDR است که مخفف Double Data rate است. وقتی ما یک DRAM سنکرون داریم

به صورت سنتی در هر کلاک یک کاری انجام می‌ده DDR می‌گه حالا که کلاک هست در هر دو تا لبه کار انجام بدیم

(هم لبه بالا رونده هم پایین رونده بنابراین در هر کلاک دو تا کار انجام می‌دهیم) یعنی نرخ ارسال دیتا دو برابر است

- در هر DRAM Device ما به جای اینکه یک بانک حافظه بزاریم multiple بانک بزاریم

اسلاید ۱۲:

یک مروری بر تکنولوژی مختلف DDR (از روی خود جدول بخون)

اسلاید ۱۳:

در این جدول مروری بر تکنولوژی DDR با توجه به بهینای باند

در این جدول I/O clock rate یعنی کلاکی که پردازنده با این RAM می‌خواد تعامل کند یعنی کلاک Bus

در این جدول Transfer دو برابر Clock rate هست چون هر دو در هر کلاک دو تا می‌فرستد

پسوند چهار DIMM یعنی کل ماژول حافظه (چیزی که روی ماژول حافظه نصب می‌شه) که بهینای باند را نشان می‌دهد،

کل دیتایی که در واحد زمان می‌تونه خارج بشه را نشون می‌ده

اسلاید ۱۴:

DDRها:

DDR2 نسبت به DDR1 توانش از ۲/۵ ولت شد ۱/۸ ولت یعنی ماشین مصرف انرژی‌اش کمتر شد clock rate هم شد ۲۶۶MHz و

333MHz و ۴۰۰MHz که در DDR1، ۱۳۳ تا بود

DDR3، ۱/۸ ولت شد ۱/۵ ولت و فرکانس ۸۰۰MHz

DDR4 شد ۱/۲ تا ۱ و فرکانس ۱۳۳۳MHz

Scanned by CamScanner

ما GDDR5 هم داریم که برای کارت گرافیک حافظه استفاده می‌شه و تکنولوژی همون DDR3 است

Year:..... Month:..... Day:.....

اسلاید ۵:

((کار با حافظه‌ها و مدیریت و چیدمان آنها))

- وقتی با چند level داریم یک level نزدیک به حافظه کوچکتر است تمام Access ها و Request هایی که پردازنده به حافظه میده در L1 می‌بینیم اگر در L1 داده باشه cache hit رخ داده اگر نباشه cache miss رخ داده

- اگر cache miss رخ داده می‌ره سراغ طبقه بعدی حافظه و داده را درخواست می‌دیم کپی‌اش هر طبقه

- اگر داده را داشت می‌ده اگر نداد می‌ره سراغ طبقه بالاتر

زمانی که پردازنده درخواست یک Data را داده بوده بجا داده‌هایی که در کش نیست و می‌ریم از بالاتر بیاریم برای آن داده‌ها، داده‌های دور و بر را هم میاریم (یک بلاک داده‌ها می‌آوریم) که یک line کش را پر می‌کند

این کار به دلیل مجاورت مکانی در درخواست‌های پردازنده است یعنی علت درخواست‌های بعدی همین داده‌های اطراف درخواست قبلی باشد

بلوک را می‌دادیم در یک set کش که براساس آدرس حافظه تعیین می‌شود که در کدام set برود

اسلاید ۱۶:

در رابطه سؤالات این اسلاید جلسه بعد صحبت می‌کنیم

Scanned by CamScanner

page frame بلوک را با علامت dirty که در بلوک حافظه نوشته می‌شود مارک می‌کنند تا بدانند حافظه تغییر کرده

Year: Month: Day:

اسلاید ۲ و ۳

ما می‌توانیم ساختار page table را به صورت چندلایه توسعه داده تا حافظه کمتر شود

ویدئو دوم (Lec3-B)

اسلاید ۳: یک کش از طرفی بلوک تشکیل شده که براساس تقسیم حافظه به صورت ۳۲B، ۶۴B و ... باشد. ما این بلوک‌ها را در n مجموعه قرار می‌دهیم که در هر مجموعه K تا بلاک وجود دارد که به این ساختار کش k-way set associative یا کش انجمنی، اگر K=1 در هر set یک بلاک وجود دارد و تعداد set‌ها با تعداد بلاک‌ها مساوی است

ساختارهای کش سه حالت دارد:

1) Direct

2) k-way

3) Fully

نوشتن در کش سه سیاست دارد:

1) write-through

2) write back

3) معمولا کش‌ها یک write buffer پشتیبانی می‌کنند برای اینکه بتوانند عملیات write را در background انجام بدهند و پردازنده را برای عملیات write معطل نداریم یعنی وقتی پردازنده چیزی را تحویل داد برای نوشتن آن چیزی به محض تحویل داده شدن کش بلکه و سریع می‌خواهد از ما تحویل بگیرد

اسلاید ۴:

ساختار روش Direct mapped:

یک کش داریم با ۴ بلاک، نگاشت ما به این صورت است که بلاک 0 حافظه به 0 کش می‌رود، 1 حافظه به یک کش می‌رود، 2 به 2 و 3 به 3 می‌رود، بلاک 4 حافظه به 0 کش می‌رود، بلاک 5 به 1 کش می‌رود و ...

هر کش را تقسیم به ۴ کنیم با باقیمانده‌اش می‌شه محل قرارگیریش در کش

در دنیای باینری ما نیازی نداریم تقسیم به ۴ را انجام بدهیم، دو بیت کم ارزش هر آدرس را نگاه کنیم آن محل قرارگیری را

Scanned by CamScanner

کش را نشان می دهد

Year:..... Month:..... Day:.....

چند بلاک می توان در یک خانه از کش قرار بگیرن ، ما خانه ی کش را با کدوم بلاک پر کنیم؟ اونی که بهش احتیاج داریم و یا در آینده نزدیک به آن احتیاج داریم

از کجا بدونیم کدوم بلاک در کش هست؟

اسلاید ۵:

مثال روش direct mapped ←

یک کش ۲KB که بلاک هاش ۳۲B باشد ساختارش در این شکل هست

بیت valid می گه آیا این بلاک داده معتبری در آن قرار دارد (پر است یا خالی) ، Tag می گه اطلاعات در یک خانه کش n فرض می تونن در آن قرار بگیرن Tag مشخص می کنه کدوم یک از خانه های حافظه اینجا قرار گرفتند

وجه اشتراک بلاک های کش Index cache است و وجه تمایزشان cache Tag است

وجه اشتراک آنها را به یک خانه مشخص از کش می برد وجه تمایز به ما این امکان را می دهد که مشخص می کنه که با هم متفاوت هستند و از کدام یک از خانه های حافظه آمده اند . offset مشخص می کنه وقتی یک بلاک در کش وجود دارد کدوم قسمت را به پردازنده بفرستیم

کش ما 2^N بایت است یعنی اینجا $2KB = 2^{11}$ یعنی N ما ۱۱ است

اندازه بلاک ما

۳۲ منهای N بیت پر ارزش آدرس ما می شه قسمت Tag ما (Tag = ۳۲ − N)

مابقی بلاک کش = 2^N که اینجا $2^5 = 32$ پس ۵ = N است پس ۵ بیت پایین آدرس (اینجا ۵ بیت پایین) می گوید کدام بایت را به پردازنده بدهیم (offset)

در مثال عددی که خودش زده از قسمت cache Index (۶ بیت از بیت ۵ تا ۱۱ از سمت راست) می فهمیم که به کدوم خانه کش بره

اسلاید ۶:

یک مثال ساده :

cach ما ۱۲۸ تا بلاک داره حافظه ۴۰۹۶ تا بلاک داره در این صورت ۷ بیت آدرس به عنوان index انتخاب میشه

۵ بیت به عنوان tag ، بلاک های ما ۱۶ بایتی هستند پس ۴ بیت اول آدرس را دور می ریزیم (یعنی به عنوان offset می زاریم کنار) ، کشه ۱۲۸ بلاک که ۷ بیت نیاز داره و آنچه می ماند به عنوان tag در نظر گرفته می شه

مزیت Direct :

سادگی است چون فقط یک انتخاب داریم برای این که دیگر از کشه

فقط یک خانه جدید می آید فقط یک خل ترانسفری داده پس هر کی در آن خل هست باید بره بیرون

Scanned by CamScanner

Year:..... Month:..... Day:.....

اسلاید ۷:

مشکل direct ← کش ما ۱۲۸ تا بلاک دارد، یک مقطعه کد داریم که می‌خواد مثلاً با به دو تا خونه حافظه access بکنه که این دو تا خونه فاصلشون از هم ۱۲۸ بلاک است. به یک بلاک access می‌کند و چون جدید کش نیست آن را می‌آورد و در کش قرار می‌دهد، درخواست بعدی به خونه‌ای از حافظه است که در فاصله ۱۲۸ از قبلی قرار دارد چون تعداد بلاک‌های cache ۱۲۸ است پس بلاک‌ها با فاصله ۱۲۸ به یک خانه اشاره می‌کنند پس بلاک دوم می‌آید چون جایی که اولی اومده بود پس اولی را می‌اندازیم دور که دومی قرار بگیره، حالا دوباره باید اولی را بیاریم دومی را بیندازیم بیرون و به صورت مکرر این انجام می‌شه و فقط سر یک بلاک دعوا است و ۱۲۷ تا کاملاً بیکار هستند

اسلاید ۸:

برای حل مشکل direct کش Fully پیشنهاد می‌شه که به که هر بلاک از حافظه هر جا کش می‌خواد بره بنابراین ما ۱۲۸ بلاک داریم ولی index نداریم و هیچ کدوم از بلاک‌ها آدرس ندارند پس دیگه مثل قبل ۷ بیت برای index نداریم و آن به tag اضافه می‌شه و tag می‌شه ۱۲ بیت پس وقتی آدرس‌ها قسمت tag است (۱۲ بیت بالای آدرس) و ۴ بیت آدرس دهی داخلی بلاک است که فعلاً با جاش کاری نداریم

مزیت Fully:

- در حافظه چون هیچ قیدی نداریم هر کس هر جایی می‌خواد می‌ره و تا زمانی که وجود دارد می‌آید و می‌تونیم حداکثر استفاده از بلاک‌ها را بکنیم

- چون انتخاب‌های مختلفی برای محل قرارگیری داریم پس باید تصمیم بگیریم که کی بره بیرون که چند تا الگوریتم برای این روش داریم و در این روش هیچ ارتباطی بین شماره بلاک حافظه با جایی از کش که می‌خواد قرار بگیره وجود ندارد

مشکل Fully:

وقتی مشتون باز باشه و قدرت تصمیم‌گیری داشته باشیم باید مدار تصمیم‌گیری بزاریم که مدار پیچیده‌ای است و ما وقتی یک بلاک را احتیاج داریم باید کل کش دنبال آن بلاک بگردیم

اسلاید ۹:

برای حل مشکلات سراغ Associative set می‌رویم

در این روش هر n تا بلاک را در یک set می‌ذاریم و بلاک‌هایی که از حافظه به کش می‌آیند تقسیم می‌کنیم به به کدوم set بروند و بلاک‌هایی که در داخل set هستند هر جا که دلمون بخواد می‌تونیم برداریم پس یک قید

Scanned by CamScanner

وجود دارد و این است که بلاک از حافظه mapped می‌شه به یک set و وقتی نگاشت شد دیگه هیچ قیدی وجود نداره
در کجا قرار بگیره

Year:..... Month:..... Day:.....

همون مثال قبلی را در نظر بگیریم که set ها هر کدوم به دو بلاک تقسیم شدن (هر set دو تا بلاک داره)، در اینجا Tag
می‌شه وجه تمایز بلاک هایی که به یک set نگاشت می‌شن

تعداد set ها ۶۴ تا است پس ۶ بیت نیاز داریم برای آدرس دهی به set ها هر چی باقی موند برای tag است
ما می‌تونیم به جای این که ۶۴ تا بلاک دو تایی داشته باشیم ۳۲ تا بلاک ۴ تایی داشته باشیم اون وقت
بلاک های ۰، ۳۲، ۶۴، ۱۲۸ به set صفر می‌رن و ... در این صورت تعداد set می‌شه ۵ بیت و
tag ما می‌شه ۷ بیت

روش k-way ترکیبی از direct و Fully است که هم هزینه‌ای که سر یک بلاک به وجود می‌آمد را
تا حدی کم می‌کنه هم هزینه‌ای که سر روش Fully به وجود می‌آمد را کمتر می‌کنه

اسلاید ۱۰:

یک مثالی از روش set associative ←

روش 2-way است که در هر ردیف یک set است که در هر set دو تا بلاک وجود داره و هر کدوم یک قسمت Tag دارن
به اضافه یک قسمت بیت valid وقتی یک آدرس می‌آید از قسمت Index یا شماره set مشخص می‌کنیم در
کدوم ردیف قرار بگیره و در آن ردیف باید قسمت tag آدرس دیتای Request داده شده را compare
کنیم با tag های در آن ردیف قرار دارد. در این عملیات compare کردن اگر تشابهی پیدا شد اونی که مشابهش
بوده بلاک مناسبش انتخاب می‌شه و می‌ره بیرون و هر چی compare ها با هم OR می‌شن که اگر 1 شد به دستم
Hit شده و اگر صفر شد یعنی miss شده و بلوکی که دنبالش می‌گشتیم پیدا نشده پس باید برم از حافظه بیرون
داریم و بیاریم اینجا بزاریم

در این روش هر set کش n تا مدخل داره و این n تا به صورت موازی با هم کار می‌کنن و tag هاشون با tag
ورودی compare می‌شه و هر کدوم از دیتاها که خواستیم انتخاب می‌شه
در روش 2-way ما از روی بخش Index آدرس set را انتخاب می‌کنیم و بعد دو تا tag را با هم
compare می‌کنیم و اونی که tag تبسیه بود دیتاش رو می‌دیم بیرون
برای این موازی compare می‌کنیم که performance بیشتری بگیریم

Scanned by CamScanner

از نظایب set associative ←

Year:..... Month:..... Day:.....

۱) اینکه n تا مقایسه باید انجام بدیم ولی در direct یک مقایسه انجام می‌دیم

۲) اون حالتی بالایی که سرعت را قرار گرفته که دیتا را از کدوم طرف بفرستیم باعث تاخیر می‌شه

۳) چون نمی‌دونیم دیتا در کدوم way است باید صبر کنیم tag تکلیف را مشخص کند بعد بفرستیم از کدوم way بیوسم

در حالی که در روش direct همزمان با tag ها دیتا را هم می‌خونیم اگر hit شده بود که خوبه هم اگر miss شده بود چیزی که خوندیم را می‌اندازیم بیرون و در main memory دنبالش می‌گردیم پس

روش direct سریع‌تر است نسبت به k-way

پس این که قبل از این که بریم بدونیم دیتا hit می‌شه یا miss می‌شه بلاک ما آماده است از مزایای

direct است حتی این امکان وجود دارد که همزمان که کار compare میشه دیتا را بدهیم به پردازنده

اگر دیدیم miss شد به پردازنده نگیم دیتا می‌گردادم بعد از بیرون تا دیتای اصلی را برات بیارم

اسلاید ۱۲:

چند بایت در هر set هست

size cache = Blocke size × number of set × set associative

associative ← تعداد set ها ← سایز بلاک

اگر به حافظه یک آدرس n بیتی به صورت شکل اسلاید ۱۲ تقسیم شده حالا از ما بلاک سایز و تعداد set و

چند بیت tag می‌خواد و ... که فرمول های این اسلاید این اطلاعات را به ما میده

اسلاید ۱۳:

یک مثال ← از جایگزینی بلاک شماره ۱۲ حافظه در یک کش با ۸ بلاک بجای کش می‌ره با فرض این که کش

Fully ، direct ، 2-way باشد

direct) برای این که بدونیم کجا می‌ره ۱۲ را به تعداد set ها تقسیم می‌کنیم باقیمانده محل را به ما نشان می‌دهد $12 \div 8 = 4$

پس در خونه ۴ کش می‌شینه

2-way) ۸ تا بلاک داریم هر ۲ بلاک یک set پس ۴ تا set داریم پس باقیمانده ۱۲ به ۴ را حساب می‌کنیم $12 \div 4 = 0$

پس می‌ره در set شماره صفر حافظه

Fully) بلاک شماره ۱۲ در هر ۸ تا بلاک بر اساس سیاست جایگزینی می‌تونه بره

اسلاید ۱۴:

چه جوری بلاک را در کش پیدا کنیم؟

هر بلاک یک tag داره و کافی است قسمت tag (بالای آدرس) را با tag مورد نظر مقایسه کنیم

Scanned by CamScanner

- هرچقدر که associativity بالا می‌ره و به سمت Fully می‌ریم:
طول قسمت index کم می‌شه و در قسمت index کوچک شبیه tag ها طولانی‌تر می‌شه

Year:..... Month:..... Day:.....

اسلاید ۱۵:

زمانی که یک miss رخ میده ما بلاک را کجا بذاریم؟

- در direct که ما حق انتخاب نداریم

- در Fully و set ما سیاست‌های جایگزینی داریم

* رندم یکی را بندازیم بیرون

* سیاست LRU: بلاک‌هایی را انتخاب می‌کنیم که مدت زیادی مورد رجوع قرار نگرفته

* FIFO: هر کی زودتر آمده زودتر هم بره بیرون

* LFU: هر کی که تعداد دفعات کمتری استفاده شده

ویدئو سوم (Lec3-c)

اسلاید ۴:

«خواندن و نوشتن از حافظه»

واقعیتی که وجود دارد این است که تعداد دفعات خواندن از نوشتن بیشتر است یعنی پردازنده درخواست‌های Read بیشتری نسبت به write دارد. در یک برنامه که برای پردازنده MIPS Run شده آمارها نشون میده که درخواست‌های Read و write پردازنده ۱۰ درصد از کل instruction‌ها بوده (store instruction) و ۳۷٪ از instruction‌ها load بوده و مجموعاً ۴۷٪ از دستورات یک برنامه دستورات خواندن و نوشتن بوده.

- اگر بخواهیم درصد بگیریم ۷٪ از کل ترافیک ما ترافیک مربوط به نوشتن است $\left(\frac{10\%}{10+37+10} = 0/7\right)$

- اگر اون ۱۰۰ را در مخرج را حذف کنیم $\frac{10\%}{37\% + 10\%} = 21\%$ یعنی ۲۱٪ از ترافیک ما مربوط به نوشتن است

- Request‌های مربوط به پردازنده فقط مربوط به نوشتن و خواندن دیتا نیست پس دیگه چی می‌خواد تبادل کنه؟

هر کدام دیتا یک instruction دارد پردازنده قبل از اینکه دیتایی بخواد بنویسه یا بخونه یک دستور خوانده پس هر خواندن و نوشتن دیتایی به طبع یک دستوری است که داره اجرا می‌شه و وقتی که operand من را بیاد می‌خوام یک کاری انجام بدن یا دستوری که من این کار را انجام دادم بریزمش در حافظه بنابراین عامل دیگری که نسبت Read به write را افزایش می‌دهد این است که علاوه بر دیتا، این که

Scanned by CamScanner

دیتا را می‌نویسیم یا می‌خونیم instruction ها را متفاوت می‌خونیم. ما n تا دستور می‌خوریم که ۰ از آنجا می‌خوان یک دیتا هم بنویسن و ۳۷ از همین ۱ از این n دستور که خوانده شدند می‌خواهند یک دیتا هم بخوان

Year: Month: Day:

پس از این ۰۰ از تمام instruction هایی است که خوانده می‌شه که از این ۰۰ از، ۳۷ از دیتا خوانده می‌شه و ۱ از هم دیتا نوشته می‌شه (در درخواست ها Read از write خیلی بیشتر است)

- وقتی پردازنده همین اینو هست از حافظه درخواست می‌خواد بده و حافظه خیلی کندتر از پردازنده است و همه را کاهش سعی می‌کند اولویت را به Read بده تا تعامل با پردازنده ایجاد نشه

- در کش وقتی درخواستی می‌آید اول آدرس آن را می‌بینیم و tag آن نگاه می‌کنیم و با tag های cache مقایسه می‌کنیم اگر hit رخ داد می‌ریم دیتای مقابلش را می‌خوانیم و تحویل پردازنده می‌دهیم. بعد از این که یک tag مشخص شد می‌بینیم دیتا را درست خوانده بودیم هزینه‌ی خواندن اضافه را می‌پردازیم و دیتا را هم می‌ریزیم دور

- این سیاست در پردازنده هایی که Embedded processors هستند یا می‌خواهیم این هزینه های مصرف انرژی را هم ممکن است کار زیادی نداشته باشد چون در Embedded process ها خیلی وقت ها انرژی بر performance اولویت دارد

- این که ما هر زمان با مقایسه tag برسیم سراغ بلوک داده و بخش Access بکنیم فقط در زمان read می‌توانیم جواب بگیریم و در زمان write نمی‌توانیم همچین کاری بکنیم در write اول باید یک tag مشخص شه و بلوک بلاک مقصد نوشتن است بلد بریم در آنجا بنویسیم

اسلاید ۳۵

مثال مهم: دیتایی که پردازنده به ما تحویل داده است وقتی می‌خواهیم در کش بنویسیم اگر آن بلاک در کش پیدا نشد چه باید کرد؟

پردازنده به کش می‌گه در فلان آدرس بنویس، کش می‌آید tag ها را مقایسه می‌کند اگر hit رخ داد write را انجام میده اگر بلوک وجود داشت در کش برای write کردن دو تا سیاست وجود داره:

1) write through ← این سیاست می‌گه وقتی پردازنده بلاکی به کش تحویل داد این بلاک باید در کش نوشته شود همزمان با اون این بلاک باید در حافظه‌ی سطح بعدی هم نوشته شود چون کش یک فضای موقت برای دیتاست پس باید در حافظه سطح بعدی هم نوشته شود تا consistency کش حفظ شود یعنی دیتا اگر از کش چند نسخه وجود دارد اون دیتاها consistence باشند و value یکسان داشته باشند

Scanned by CamScanner

مثلا آدرس ۱۰۰ در cache و حافظه value یکسان داشته باشند

Year:..... Month:..... Day:.....

این سیاست دو مسئله در مورد پردازنده های multi core هم هست

زمانی که ما یک کش داریم در یک core یک کش دیگه داریم و ... یک دیتایی را از حافظه آوردیم در کش یکی از core ها قرار دادیم و دیتای اول core ما عوض کردیم، core دیگری به کش خودش میگه این دیتا را می خوام اون کش می گه من دیتا را ندارم و باید بریم از حافظه بیارم و وقتی میره از حافظه بیاره دیتایی را می آورد که consistence نیست با دیتایی که در اول یکی core هست پس سیاست write through این مشکل را حل می کند

در مقابل سیاست write back هست ← این سیاست می گه اگر دیتا را در کش پیدا کردی پردازنده دیتای جدیدی که داده اوردی ~~[illegible]~~ قبلی ها آپدیت کن و هیچ جای دیگه هم نیاز نیست دیتا را آپدیت کنی

برای این سیاست لازم است مشخص کنیم کدوم بلاک های کش تغییر کردند و کدوم تغییر نکردند و کدوم ها را باید برگردونیم به حافظه اگر خواستیم آن ها را بیرون بندازیم که در این صورت نیاز به بیت dirty داریم

فایده این سیاست اینه که ترافیک ما به main memory یا حافظه را کم می کند (منظور حافظه بعد سطح کش هست) چه جوری ترافیک را کم می کند؟

به ازای هر write مجبور نیستیم دیتا را در حافظه بلافاصله هم بنویسیم و تعداد ترانزکش های بین کش level 1 و level 2 صورت می گیرد کمتر است و bus که بین این دو است ترافیک کمتری دارد و این bus وقتی در یک multi core باشد که در multi core شلوغ بودن bus گلوگاه ایجاد می کند بهتر عمل می کند

و write through چون ترافیک زیادی ایجاد می کند خیلی مجبور نیست

ما به عنوان معمار کامپیوتر باید ببینیم در بین سطوح کش از کدوم سیاست استفاده کنیم بهتره

اسلاید ۶:

حالا اگر miss رخ داد چه write allocation ایجاد می شه که دو تا سیاست داره:

۱) سیاست write Allocate ← این سیاست می گه اگر دیتا را در کش پیدا نکردیم و دیتا را می خواهیم آپدیت کنیم اول باید از main memory بلاک را بیاریم در کش بندازیم و بعد دیتایی که پردازنده به ما تحویل داده و یک قسمت از این بلاک هست را آن جا می نویسیم و آپدیت می کنیم (الان بلوک در کش وجود داره پس دیتایی که پردازنده ما تحویل داده را در کش می نویسیم) معمولا در سیاست write back از write Allocate استفاده می کنیم

Scanned by CamScanner

۲) سیاست No write Allocate ← این سیاست می‌گه حالا که دیتا در کش وجود نداره بیخیال کش نشو و دیتا رو توی همون جایی که هست بنویس مثلاً در L_1 هست پس L_1 را می‌زاریم کنار و در L_2 می‌نویسیم اگر در L_1 و L_2 و L_3 نباشه بالأخره در main memory هست پس بریم در اونجا بنویسیم

Year:..... Month:..... Day:.....

این سیاست معمولاً برای write through استفاده می‌شه و خوبش اینه که وقتی write miss داریم دیگه مجبور نیستیم هزینه آوردن یک بلوک به کش را بپردازیم. در write through وقتی دیتایی رو میده به هر حال باید در حافظه اصلی هم بنویسیم حالا که در اونجا می‌نویسیم دیگه نوشتن در کش هزینه داره پس بهتره کش را دور بزنیم

در سیاست write back برای این از write Allocate استفاده می‌کنیم که write back می‌گه اگر دیتا را در کش داشتیم سراغ حافظه اصلی نرفتیم پس بریم دیتا را از حافظه بیاریم در کش بنویسیم بعد آپدیت کنیم خب یعنی هزینه نوشتن دیتا در حافظه را هر چقدر جا بیندازیم با هزینه آوردن دیتا از حافظه بعد نوشتن در کش را انجام بدیم و بیت dirty را یک کنیم و به هر حال این نوشتن در حافظه که الان از کش در نمیتم بعدش باید انجام بدیم پس write Allocate یک read در کش داره یک نوشتن از کش به حافظه بعدش داره

پس مزیتی که داره اینه که با فرض این که Locality داریم در دسترسی به حافظه وقتی پردازنده یک دیتا به صاحبده و ما به محض آن یک بلوک میاریم در کش می‌نویسیم و آن بلوک را آپدیت می‌کنیم امیدواریم در آینده نزدیک پردازنده دیتاهای بعدی آن بلوک را نخیر به حافظه و دیگه به ازای تک تک دیتاها مجبور نیستیم write انجام بدیم و از write های تکراری جلوگیری می‌کنه

- مسئله بعدی بحث write buffer هست که معمولاً کش‌ها برای این که تأثیر پردازنده را بعد از نوشتن بکش یک write buffer دارند

write buffer چیه؟ یک buffer است که در آن write انجام می‌شه، فایده‌ای که داره اینه که زمانی که ما یک بلوک و دیتایی را می‌خواهیم در یک جای دیگر بنویسیم مثلاً در L_1 هست در L_2 می‌خوایم بنویسیم برای این که پردازنده عقب نشود write buffer میاد دیتاهایی که به جایی خواد write شبه تحویل بانک می‌دهم پردازنده می‌ره و ~~[illegible]~~ به شبه کار ادامه می‌دهد و مثلاً L_1 خودش می‌ره و از بانک می‌خوره

Scanned by CamScanner

اسلاید ۷:

write buffer به این صورت عمل می‌کند که اگر سیاست write though باشد پردازنده هر زمانی که می‌خواد در کش بنویسه در بافر هم می‌نویسه و memory controler همیشه از buffer می‌خونه و در DRAM می‌نویسه و همین نوشتن دیتا به حافظه تداخلی با همین خوندن نداره و موقع خوندن معطل نمی‌شیم

ساختار write buffer، FIFO است، سایزش می‌تونه به اندازه ۴ بلاک، ۸ بلاک و بزرگتر هم باشد

و این write buffer می‌تونه عملاً کل Latency مارو به خاطر write ها با نوشتن در پردازنده و کش نسبت به حافظه‌های بعد از خودشون فقط مقطعی خواندن دارن و مقطعی نوشتن اصلاً ندارن و در buffer می‌ریزن هر چی می‌خوان بنویسن اما این مشروط به این است که حجم دیتایی که در واحد زمان می‌نویسیم از Latency عدد خیلی کوچک‌تری باشه یعنی ترافیک نوشتن باید نسبت به تأخیر نوشتن کم باشه

اگر ترافیک نوشتن زیاد باشه کار خراب می‌شه و write buffer اشباع می‌شه و دیگه پر می‌شه و دیگه نمی‌تونه جلوی مقطعی بایستد و مجبوره کش بالاتر صبر کنه تا چیزایی که قبلاً داده در حافظه بنویسه و جا خالی بشه

یکی از کارهایی که عامل کامپیوتر باید انجام بده اینه که بینه بیشتر برنامه‌ها در L1 و L2 ، L2 و L3 و ... چقدر باشه

و این علاوه بر این که تأثیر سرعت پردازنده است تأثیر اجرای برنامه هم هست (چون یکسری برنامه‌ها زیاد در write می‌کنن و کار دارن)

اسلاید ۸:

یک جمع‌بندی راجع read و write در کش

	hit	miss
read →	دیتا را از کش می‌خونیم و به پردازنده می‌دیم	یک بلوک از حافظه بیاریم و در کش جا براش آزاد کنیم و دیتا را تحویل پردازنده بدیم
write through →	دیتا را هم در کش هم در حافظه می‌نویسیم	دیتا را فقط در حافظه می‌نویسیم اگر سیاست no write Alloc باشد اگر write Alloc باشد هم در حافظه هم در کش می‌نویسیم
write back →	اگر سیاست WA باشد در زمان hit بلوک در کش وجود داره و آپدیت می‌کنیم و dirty را یک می‌کنیم	اگر سیاست WA باشد یک بلاک از کش را خالی می‌کنیم اول دیتایی که می‌خوای write بشه بنویسی را می‌ذاریم در کش و بعد آپدیت می‌کنیم

Scanned by CamScanner

هر چی miss rate کمتر باشه بهتره

اسلاید ۹:

- یک پارامتر درس داریم به نام miss rate (نرخ فقدان) ← درصد Access هایی که به کش می‌رسه و وجود نداره (درصدی که miss رخ داده)

- miss که در کش رخ میده نتیجه سه پدیده است:

۱) Compulsory ← به این دلیل رخ میده که پردازنده درخواست داده به خانه‌ای از حافظه که تا حالا مجبور درخواست بهش نداده بوده

۲) Capacity ← زمانی رخ میده که کش به خاطر این که ظرفیتش محدود است نتونسته دیتایی که قبلا آورده و می‌دونه پردازنده دوباره میخواد را در خودش نگه داره

۳) Conflict ← به دلیل رقابت در بلاک به سر یک خونه کش است برای حل این مشکل Associativity را بالا باید ببریم

اسلاید ۱۰:

$$\text{miss per instruction} = \text{miss rate} \times \overbrace{\text{memory Access}}^{\text{تعداد درخواست‌های پردازنده به حافظه}} = \text{miss rate} \times \frac{\text{memory Access}}{\text{Instruction Count}}$$

Instruction Count ← تعداد Instruction هایی که پردازنده در آن بازه زمانی اجرا کرده

برای این میریم سراغ miss per Instruction که miss rate خودش به تنهایی نمی‌تونه چیز خوبی نیست

چون miss rate نماینده‌ی خوبی برای سنجش performance نیست مثلا دو تا برنامه داریم یکی miss rate = ۵۰٪ یکی miss rate = ۲۰٪ اگر از ما بپرسند کدوم performance بالاتری داره و کش در این کدوم بهتر عمل کرده شاید بگیم کش برای اون که miss rate کمتری داشته بهتر است اما واقعیت چیز دیگری است

هر دو برنامه ۱۰۰۰ تا Instruction دارن اونی که miss rate = ۵۰٪ است از ۱۰۰۰ تا Instruction ۹۰۰ تا از آنها Computation های Instruction بود و ۱۰۰ تا از آنها Instruction های Access به حافظه بوده پس یعنی از اون ۱۰۰ تا Instruction ، ۵۰ تا miss شده ۵۰ تا hit شده پس از کل ۱۰۰۰ تا ۵۰ تا عمل حافظه بودن و ۹۵۰ تا عمل نبودن. اون یکی برنامه ۵۰۰ تا دسترسی به حافظه داشته ۵۰۰ تا دسترسی به حافظه نداشته پس وقتی می‌گیم miss rate = ۲۰٪ است از ۵۰۰ تا که دسترسی به حافظه داشتن miss شدن یعنی ۱۰۰ تا پس از کل ۱۰۰۰ تا ۱۰۰ تا عمل حافظه شدن ۹۰۰ تا عمل نشدن پس برنامه با miss rate = ۵۰٪ بهتر است

Scanned by CamScanner

Average memory access time = Hit time + miss rate × miss penalty

برنامه هایی که دسترسی به حافظه داشتن به طور متوسط چقدر فعال شدن — زمانی که برای hit شدن خرج می‌شه — تعداد دفعات miss (درصد miss) — هزینه زمانی به ازای هر miss

Year:..... Month:..... Day:.....

نکته: miss rate ، miss per Instruction ، Average memory access

~~میشه~~ باز هم بسته زمانی می‌تواند میزان اثرگذاری miss بر performance را خوب نشون بدن چون پردازنده خیلی زرنگتر از این است که خودش را معطل miss کند یا خودش را معطل حافظه کند تازه پردازنده به صورت حدس و گمانی هم ~~پردازنده~~ برنامه ها را اجرا می‌کند یعنی پردازنده زمانی که به یک Instruction برمیخوره و توی Instruction به جایی متوقف شده و به هر دلیلی از جمله این که دیتا در کش نبوده و miss شده ، پردازنده این Instruction را می‌ذاره کنار و به Instruction های بعدی می‌ره بیاید جلو تا را اجرا بکنم شاید بتونم کارتون را راه بندازم و این تکنیک ها باعث می‌شه اثر miss ها روی performance به حداقل برده و بتواند به راحتی این سه فرمول را به performance .

اسلاید 11:

«Replacement policy»

با توجه به این که سایز کش محدود است مجبوریم هر وقت کش پر شد و خواستیم دیتای جدید بیاریم باید ببینیم چه کسی را بندازیم بیرون ، اونی را می‌ندازیم که امید داریم در آینده ی نزدیک به آن احتیاج نداشته باشیم و وقت هایی که از آن ها استفاده نکرده ایم

ویدئو چهارم (Lec3-D-1)

اسلاید 12:

«ادامه ی مبحث Replacement policy»

Replacement policy یک تصمیم مهم باید بگیرد و آن این است که چه کسی به عنوان victim (قربانی) انتخاب شود یعنی زمانی که miss رخ می‌دهد چه کسی را بیرون بیاندازیم تا بلاک جدید جای او قرار بگیره

سیاست های متنوعی برای این تصمیم گیری وجود داره

1) FIFO: هر بلوکی که زودتر اومده زودتر هم برود بیرون (مثلا بلوک 5 ام که اومده بلوک اول برود بیرون)

مشکلی که FIFO داره اینه که سناریوهایی وجود دارد که وقتی associativity را بالا می‌بریم

Scanned by CamScanner

miss rate ها نه تنها کم نمی‌شود بلکه بالا هم می‌رود به همین دلیل FIFO سیاست مطلوبی نیست

Year:..... Month:..... Day:.....

۲) LRU: سیاست مطلوبی است اما پیچیدگی‌های سخت افزاری زیاد دارد

و برای ساده‌اش pseudo-LRU است

۳) LFU

۴) NRU

۵) Random

۶) optimal ← آن سیاستی است که همه‌ی سیاست‌ها خودشون را می‌خوان به اون نزدیک کنند ولی

سیاستی است که وجود خارجی ندارد اما می‌تواند معیاری برای ارزیابی دیگران باشد

درش بهترین گزینه برای اخراج از بین بلوک‌های یک set بلوکی است که در آینده‌ی دورتری قرار است

Request شود یعنی اگر ما آینده را بدانیم و از بلاک‌هایی که درش هستند بدونیم در آینده کدوم از آنها تا مدتی قرار نیست

Access شود کسی را باید بیاریم بیرون که دورتر است و آینده دورتری می‌خواد Access شه و سیاست optimal

چون باید از آینده خبر داشته باشد وجود خارجی ندارد اما زمانی که ما کاری را براساس یکی از سیاست‌ها انجام دادیم

باید برگردیم به عقب و مقایسش کنیم با سیاست optimal که اگر این تقسیم optimal را می‌زد چی می‌شد و

حالا که این تقسیم را زد چقدر ضعیف miss rate شده و چقدر performance داشتیم

اسلاید ۶:

سیاست LRU:

می‌دارین بلوک‌هایی که بیشتر می‌خوام انتخاب کنیم برگردیم به عقب و اون که مدت دورتری Access شده

است

- اگر تعداد way های ما 2 باشد (2-way) یعنی در هر set کش ما دو تا بلوک وجود داشته باشد LRU

می‌شود معادل NRU. یک بیت برای اون 2-way کفایت می‌کند و همین یک بیت همه‌ی کارها را می‌توانیم انجام بدهیم

- اگر تعداد way های ما بیشتر از 2 باشد

یک راه برای این که بدونیم کدوم بلاک را بیاریم بیرون این است که زمان Access شدن هر بلوک را یه جا

save کنیم اما از کجا بیاریم یکجا set کنیم

مشکلی که این روش دارد اینه که ما چند بیت بذاریم برای Counter که می‌خواد بیاد در خودش ذخیره کند

Scanned by CamScanner

یک راه دیگه اینه که بلوک ها را براساس زمان Access شدنشون sort کنیم یعنی یک صف تشکیل می‌دیم از بلوک‌ها

هر بلوکی که Access شد بیاد سر صف و هر کی دیرتر Access شد بره عقب تر صف

Year:..... Month:..... Day:.....

هر وقت کسی را می‌خوایم بندازیم بیرون از ته صف بندازیم بیرون و هر کسی هم که Access می‌شه جابه‌جا می‌کنیم

به سر صف. پیاده‌سازی این روش مسئله‌دار است به هر بلاک به اندازه‌ی $\log_2^W$ بیت اضافه کنیم

مثلاً یک کش 4-way می‌شه $\log_2^4 = 2$ پس به هر بلاک ۲ بیت اضافه کنیم و ....

این بیت‌های اضافه می‌خوان تریک Access ها را نشون بدن. این بیت‌ها که بین بلاک‌های set تعداد unic دارن هر چه عدد کوچکتری داشته باشند نشان دهنده این است که این بلوک به سر صف نزدیکتر است و هر چه بزرگتر باشند نشان دهنده این است که به ته صف نزدیکتر است

اسلاید ۷:

الگوریتم LRU در این اسلاید نوشته شده

در سیاست LRU هر وقت بلاکی hit می‌شود آن n بیت که به یک بلاک اختصاص دادیم (چون $\log_2^W$) همه‌ی صفر قرار بدیم و تمام بلاک‌هایی که آن n بیتشون عددی کوچکتر از عدد اونی که الان Access شده داشتند یکی بهشون اضافه می‌کنیم (۱ اضافه یک می‌کنیم)

اگر miss رخ بدهد ما دنبال جایی که لاین خالی وجود دارد (از بیت valid اون متوجه می‌شیم) بلاک جدید را اونجا می‌ذاریم و counter اون را هم صفر می‌کنیم و بقیه counter ها را هم یکی اضافه می‌کنیم اما اگر بلوک خالی وجود ندارد اونی counter تعداد بزرگتری دارد را به عنوان victim انتخاب می‌کنیم و می‌ذاریمش بیرون و کانترش را صفر می‌کنیم و بلاک جدید را جای اون می‌ذاریم و کانتر بقیه بلوک‌ها را یکی اضافه می‌کنیم

LRU می‌خواد به جوری ادای optimal را دربیارد

اسلاید ۸:

وقتی associativity بالا می‌رود LRU سخت می‌شه و اون همه کانتر را آپدیت کردن و دنبال بزرگترین کوچکترین و به هم را صفر کردن و ... در دسترس زیاد دارد و پیچیدگی سخت افزاری دارد

سیاست ساده‌تر که نزدیک به LRU است pseudo-LRU است

در این سیاست برای n تا بلاک که در یک set وجود دارن n-1 بیت در نظر می‌گیریم (کل set

n-1 بیت دارد

Scanned by CamScanner

این سیاست به این صورت عمل می‌کند که برای یک set که در این اسلاید هر set ۴ تا بلاک دارد سه تا بیت $b_0 b_1 b_2$ در نظر می‌گیریم اگر بخش اول یعنی لاین ۰ یا ۱ Access شده b_0 را یک می‌کنیم اگر بخش دوم یعنی ۲ یا ۳ Access شده b_1 را صفر می‌کنیم بعد بین ۰ و ۱ می‌آییم ببینیم کدوم Access شده است اگر لاین ۰ بود و Access شده بود b_0 را یک می‌کنیم اگر نه b_0 را صفر می‌کنیم اگر لاین ۲ و ۳ Access شده بود b_1 را یک می‌کنیم و اگر بعد اگر لاین ۲ Access شده بود b_1 را یک می‌کنیم وگرنه b_1 را صفر می‌کنیم

اسلاید ۹:

مداری که می‌تواند این سیاست را پیاده‌سازی کند (سیاست pseudo-LRU) چه سادگی مدار همین اسلاید است و خیلی ساده‌تر از LRU است

این مدار برای 4-way طراحی شده. طبق این سیاست اگر $b_0 b_1 = 00$ باشد لاین صفر را می‌اندازیم بیرون اگر $b_0 b_1 = 01$ باشد لاین ۱ را می‌اندازیم بیرون و به همین صورت

اسلاید ۱۰:

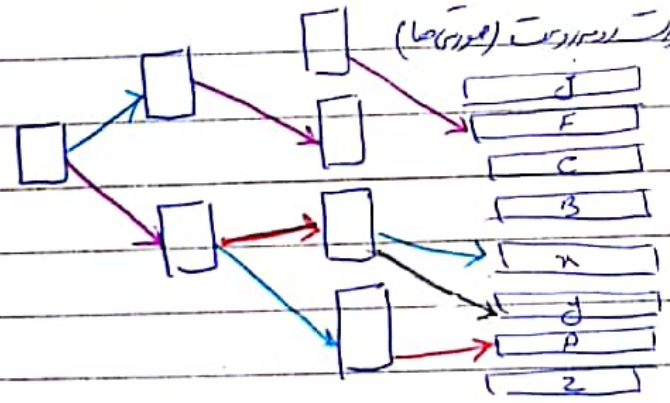
انتظار می‌رود pseudo-LRU مواقعی را نشان می‌دهد که با اون ۱۰ بیت درخت تشکیل داده نه این شکل 8-way است پس ۷ بیت داره

اسلاید ۱۱

داره به صورت جزئی شکل اسلاید قبل رو می‌ره که اول ۸ تا لاین شون می‌ره داخل هر لاین هم یه set هست مثلاً در لاین اول ل set است در لاین دوم F set است و ...

فرض کنیم Request شبیه ل و چون ل تازه Access شده اگر در آینده نزدیک miss رخ داد نباید ل را قربانی شود پس بیت‌هایی که راهنمای ما هستند باید جوری هدایت کنیم که به سمت لاین‌هایی بره که دورتر نشده دورتری

مثلاً Access شده پس فلش‌ها طوری به صورت روبرو درست (جوری‌ها)



اگر ل Access شد و الان بخواد miss رخ بده باید بفرستیم سمت بالا که victim شده و اگر اومدیم سمت x ل باید فلش رو به سمت x بزنیم که به x بره (آبی‌ها)

Scanned by CamScanner

اگر Access شد می‌شه اول مشکل‌ها (باید تغییر کنه به مشکل‌ها) اگر 2 به Access شبه میشه اول مسیر قرمز

(باید آپدیت شبه به اول قرمز) و بقیه هم به همین منوال است (حدود دقیقه ۱۰ توضیحات)

Year:..... Month:..... Day:.....

اگر از جایگزین MRU چه کسی است یعنی آخرین کسی که Access شده (جوان‌ترین) چه کسی است

باید خلاف جهت فلش‌ها حرکت کنیم (یعنی هر جا که فلش نداره می‌رویم به اول سمت)

اسلاید ۱۲:

برای هر Access که برایش داریم یک مسیری را آپدیت می‌کنیم

اسلاید ۱۳:

LRU غیر مشکلی که برای سخت افزارش داره مثل پیچیدگی و ... مشکلات دیگری هم دارد از جمله این که داده‌ها

الگوی Access برداشته به دیتاها چون به صورت streaming باشه یعنی حالت scan کردن

فضای حافظه داشته باشد و یعنی حجم انبوهی از دیتاها را بخوانیم از اونشون رد بشیم و دیگه دیگه

Access نکنیم و از اونشون بریم جلو مثل پیمایش یک ماتریس در این حالت عملاً لوکالیتی زمانی در Access

به بلاک‌ها یا دیتاها وجود ندارد چون هر دیتا یک بار Access می‌شه و می‌ریم سراغ بعدی در این حالت LRU

به چیه بدردی می‌خوره؟ LRU کل تلاشش می‌شه این است که اون لوکالیتی زمانی را ضبط کند و وقتی

لوکالیتی زمانی وجود نداشته باشد این همه هزینه هم بدردی نمی‌خوره

مشکل دیگه مشکل Trashing است یعنی آلوده کردن cache چیزهایی بیخود فرستادن درش و

چیزهای خوب را بیرون انداختن. Trashing در کش موقعی رخ میده که Reuse distance ها

از associativity ها بزرگتر باشه. Reuse distance یعنی یک بلاک Access می‌شود

Access بعدش چقدر فاصله داره از این بلاک. ۱۰ تا بلاک داریم و از بلاک ۱ تا ۱۰ Access کردیم

و از بلاک ۱ تا ۱۰ می‌رفتیم در کش بعد از این که بلاک ۱ تا ۱۰ شد اولین Access بعدی به بلاک

اول رخ میده و اینجا مشکلی رخ نمیده ولی اگر ۱۱ تا Access داشته باشیم

۱+۱۰ تا بلاک و بعد ۲+۱۰ امین Access ما چیه می‌شود. بلاک اول تا ۱۰ Access می‌شن و در کش

هستند، بلاک ۱+۱۰ ام که Access می‌شه در کش miss می‌شه و ما می‌ریم از حافظه میاریم و در set می‌ذاریم

جای بلاک اول، بلاک اول می‌ره بیرون و بعد از این ما دوباره Request داریم به بلاک اول و بلاک اول

در کش نیست و باید بریم بیاریمش. Reuse distance یعنی چقدر فاصله بین این Access به بلاک اول تا

Scanned by CamScanner

Access قبلیش که بین این دو تا Access حداقل n تا بلاک دیگه Access شدن و تمامی این بلاک ها فعلا در همین set

Year:..... Month:..... Day:.....

درحالتی که ما scan داریم و دیتاها را پشت سر هم Access می‌کنیم و Temporal locality وجود ندارد حتی اگر Temporal locality هم وجود داشته باشد اگر تعداد بلاک هایی که باید scan شود از associativity بزرگتر باشد دوباره همون مشکل پیش میاد و عملا به جوراییی دچار Trashing می‌شیم یعنی بلوک ها میاریم استفاده می‌کنیم و نمی‌تونیم مجددا استفاده کنیم ازش و دوباره باید بیاریم و دچار miss های متوالی می‌شیم و هزینه داره برای ما و کش ما نه تنها کمکی نمی‌کنه بلکه هزینه اضافی هم تحمیل می‌کنه (هزینه رفتن و بردن دیتا) و اگر کش نبود مستقیم می‌رفتیم سراغ حافظه و این همه هزینه و مشکل کش را نداشتیم

اسلاید ۱۴:

الگوریتم LIP:

این الگوریتم به این صورت است که می‌گه زمانی که اپلیکیشن ما intencive memory است یعنی اون اپلیکیشن Request هاش به حافظه زیاده یعنی دستوراتی که کار دارن با حافظه زیاد (اپلیکیشنی که computation intensive است یک دیتا را از حافظه میاره طی محاسبات انجام می‌دهد و دستورات محاسباتی از دستورات Access به حافظه بیشتر است) LRU خیلی خوب عمل نمی‌کنه به خاطر همون بحث scan و Trashing.

الگوریتم LIP می‌گه به جای این که وقتی یک miss رخ میده و بلوک جدیدی را به کش می‌آوریم وارد بلوک را می‌ذاریم سر خط به جای این که بلوکی که تازه اومده بیاید سمت جوان‌ترین بلوک چون در LRU وقتی بلوک جدیدی وارد می‌شود کانتر آن را صفر می‌ذاریم و تا کانتر آن n-1 شه اون بلوک خارج نمی‌شه که خب این احتمال زیادی است که LIP می‌گه بلوکی که الان می‌آورید ممکنه چندان به درد نخور باشه که LIP می‌گه اون بلوکی که جدید اومده را بذار پیرترین بماند اگر دوباره همین Access شد آن را جوان کنید با این تغییر اتفاقی که می‌افته اینه که جسم دیتایی که باید برداریم نمی‌شیم و این دیتا را باید به صورت مکرر برداریم نیستم از جسم حافظه برداشته است در LRU ما این دیتا را کنار می‌آوردیم و قبل از این که دوباره استفاده شن می‌ریم بیرون و هر بار Access می‌شه ولی با این سیاست یکبار دیتا را می‌آوریم بعد دیتاهای جدید می‌آوریم دیتاهای قدیمی را می‌خوایم بندازیم بیرون اول دیتاهای جدیدی که میان و وقتی جای LRU برای این

Scanned by CamScanner

Year:..... Month:..... Day:.....

نه MRU یعنی بیرون می‌موند و جوان‌ترین نمی‌شن این شانس را به بلاک‌هایی که قبلاً بودن میده که در کش بمونن و بلاک‌های جدید زودتر بیرون بیرون و حداقل یک بخش قابل توجهی از خانه توی کش بمونن و بیرون انداختن (مثل در LRU این بود که هر کی یکبار Access می‌شد LRU

خب حالا می‌گن که این متد برای حالتی که scan داشتیم و مثلاً میاد دیتا از میانه کش برمی‌داره یا اصلاً مشکل را حل می‌کنه ولی LRU را به نوبه‌ی خودش کلی دیتایی بعید است و خوب نمی‌کرد پس به وقت‌هایی LRU همه خوب باشه به وقت‌های IP نا پس بیایم اینجا ترتیب کنیم ما هی وقت‌ها بزاریم کسی که جدید میاد سر صف باشه ما هی وقت‌ها ته صف بزاریم.

تغییر دیگه‌ای که در سیاست دادن و نسخه‌های IP LIP است من اون اولی را با دومی ترکیب کنیم یعنی بیایم ما نی دومی جلو LRU عمل کنیم یا جلو LIP عمل کنیم دیتایی که میاریم را سر صف بزاریم یا ته صف. بیایم رفتار برنامه را مانیتور کنیم در لحظه و بر اساس آن چیزی که الان اتفاق می‌افته تصمیم بگیریم که LRU عمل کنیم یا LIP به چه صورت مانیتور کنیم؟ بیایم به تعداد خیلی کوچکی از set های کش را، در حد مثلاً ۳۲ تا set از هزاران set در کش را به عنوان set های نماینده در نظر بگیریم و رفتار اینجا را مانیتور کنیم مثلاً دو تا ۳۲ تا set را در نظر بگیریم این ۳۲ تا LRU رفتار بکن ۳۲ تای دیگه با LIP رفتار بکن بعد بیایم آخرین تحولاتی که در کش رخ داده miss rate بکنیم با الگوریتم miss rate هر کدوم بالاتر بود برای بقیه‌ی set ها عمل بکنیم

اسلاید ۵۱:

سیاست بعدی NRU است

این سیاست می‌گه من می‌خواهم بیایم اونی که از همه جوان‌تر است نذارم بیرون

می‌گه میاد به هر بلاک یک تک بیت می‌چسبونه که این تک بیت یا ۰ است یا ۱، هر کسی وارد کش می‌شه اون تک بیتش یک است هر وقت بلاکی Access می‌شه این بیت را صفر می‌کنیم. Replacement policy

به این صورت عمل می‌کند که می‌گه زمانی که miss رخ داد من برم از بین بلوک‌هایی که بیت‌شون یک هست یکی را برای انداختن بیرون و مهم نیست که کدوم را بندازم بیرون.

همه شرایط پیش بیاد که این تک بیت همه صفر باشند در این شرایط همه‌ی تک بیت‌ها را یک می‌کند و یکی را رندم بیرون می‌اندازد

Scanned by CamScanner

اسلاید ۱۶:

سیاست آخر LFU

Year:..... Month:..... Day:.....

از بین بلوک هایی که در کش هستند اونی رو بیرون کن که تعداد دفعات کمتری Access شده. یک counter

روی هر بلاکی داریم هر کسی هر بار Access می‌شه کانترش رو یکی اضافه می‌کنیم و وقتی می‌خواهیم کسی

را بندازیم بیرون اونی که کانتر کمتری داره می‌اندازیم بیرون. هر کی هم که جدید میاد کانترش را باید صفر کنیم

این کانتر چند بیتی باشد؟ اگر یک بیتی باشد می‌شه همون سیاست NRU پس یک بیتی بدرد نمی‌خورد

مشکلی که این سیاست را نابود می‌کند اینه که زمان را در کن در نظر نمی‌گیریم و بلوک هایی که در کش اومدن

همه با هم نمی‌رن و اونی که خیلی وقته اومده خب کانتر بیشتری داره نسبت به اون که ۵ دقیقه پیش اومده

ویدیو پنجم (Lec3-F1)

اسلاید ۴:

با فرض این که یک کش را داریم چه کارهایی انجام شده است که performance کش را بهبود بخشیم؟

۱) سایز بلوک های کش را بزرگ کنیم: با فرض این که یک کش با سایز ثابت داریم دست ما است که مشخص

کنیم بلاک هاش چند بایتی باشن و set ها چند way باشند. اگر سایز بلوک کش را بزرگ کنیم بر حسب مورد

miss های compulsory کاهش پیدا می‌کنه

miss هایی که به دلیل Request اول یک مسیر رخ می‌دهد چون تعداد بیشتری داده می‌آورد هم آورده می‌شود

با بزرگ شدن سایز بلوک ها تعداد آنها کمتر می‌شود در نتیجه حق انتخاب کمتری شده و این باعث میشه

conflict miss و capacity miss بیشتر شود، miss penalty هم زیاد می‌شود چون

بلاک که برداشته می‌شود زمان آوردن آنها هم زمان بیشتری طول می‌کشد

۲) سایز کش را افزایش دهیم:

۲) بزرگ کردن سایز بلاک خوب است اما از یه جایی به بعد نمی‌شه بزرگ کردن سایز بلاک خوب نباشه

پس باید سایز optimum برای کش را بدست بیاوریم و بگیم باید چند بایت باشه

یا سؤال بدیهی تر: می‌دونیم بلاک سایز در کش هایی که امروزه استفاده می‌کنیم 64 byte (512 bit)

به عنوان یک واحد ظرفیت باشد چرا اینکه ظرفیت 32bit یا 64bit که Data ها عدداً 4 byte و

۳۷

8 byte هستند چرا بلاک سایز کش در این بازه ها ۶۴ بایتی است؟

Scanned by CamScanner

بحث بعدی اینه که کل سایز کش را بتونیم بزرگتر کنیم و به جای این که L1 کش ۳۲k بذاریم ۱۲۸k یا ۶۴k بذاریم

Year: Month: Day:

بدیهی که هر چی کش بزرگتر شه miss rate ما کمتر می شه و دسترسی ما به سطوح بعدی کش راحت تر می شود ولی بزرگ کردن سایز کش باعث می شه hit time برود بالا

(hit time: کش زمانی که می خواد دیتا به پردازنده بده در ازای یک ریکوئست و وقتی Data را داره با آخره باید یک زمانی رو صرف کنه برای این کار hit time نسبت به miss penalty خیلی عدد ناچیزی است،

hit time تا جایی که می شه باید نزدیک شبه به فرکانس کاری پردازنده اما یک تا نون ملی وجود داره که می گه هر وقت یک حافظه بزرگتر می شه کندتر می شه یعنی L1 به ازای بزرگتر شدن باید به ازای تک تک hit ها یک کلاک اضافه به پردازنده تحمیل کنم باید بنشینم این بعضی ارزش دارد) در کنار hit time، انرژی power هم افزایش پیدا می کند و باید بنشینیم می صرفه بزرگتر کردن یا نه

به این دلیل L1 ما ۳۲k است که یک تصمیم مهندسی است چون با افزایش یا کاهش سایز آن مشکلاتی ایجاد می شه

۳) associativity کش را بالا ببریم

وقتی سایز بلاک ها و کش ثابت باشه با بالا بردن associativity تعداد set ها کم می شه و تعداد بلوک های داخل set زیاد می شه و باعث می شه conflict miss کم شه،

با بالا رفتن associativity کش complex تر می شه مدارهاش پیچیده تر می شه و access time بیشتر می شه یعنی hit time افزایش پیدا می کند و این به خاطر رهاست به اضافه ی این که عملیات بیشتری باید انجام بدم و انرژی بیشتری باید مصرف کنم

در نتیجه به این دلیل L1 ما 4-way است یا L2 ما 8-way است و L3 ما 16-way یا 32-way است که تصمیم مهندسی است تا اون مشکلات بالا ایجاد شه

۴) تعداد سطوح (level) کش را زیاد کنیم

این که چند level بذاریم باید بنشینیم نسبت سطوح مختلف از نظر access time چه با تعداد نسبت سایزی آنها چه باشد ولی در نهایت از یه جایی زیاد کردن تعداد سطوح فایده ای نداره و عملکرد به صرفه نباشه

Scanned by CamScanner

۵) اولویت بدیم به read نسبت به write حتی اولویت بدیم به read-miss نسبت به write-miss :

Year:..... Month:..... Day:.....

پردازنده می‌خواد چیزی رو از کش read کنه و این رو نداره یک چیزی رو هم می‌خواد write کنه اول اون write رو نگه می‌داریم و read-miss رو جواب می‌دیم بعد write رو انجام می‌دیم

۶) در ساختار کش به خصوص L1 و L2 ، ترجمه‌ی آدرس را دور بزنیم

در کش هم آدرس‌ها بر اساس physical address ذخیره save می‌شن ، حالا اگر پردازنده ریکوئستی به کش داد ، پردازنده داره آدرس logical تولید می‌کنه ولی کش آدرس physical داره این وسط باید آدرس logical توسط PT که کش شده از TLB به آدرس فیزیکی تولید کنه و اینجوری عملکرد پردازنده زیاد می‌شه و اگر بتونیم دور بزنیم این ترجمه رو و از طریق همون logical address بتونیم data رو در L1 پیدا کنیم و به پردازنده بدیم کار خیلی راحت‌تر می‌شه

اسلاید ۵:

در کنار اون ۶ مورد اسلاید قبل کارهایی که هم می‌تونیم بکنیم :

۱) تا جایی که می‌تونیم hit time را در کش کم کنیم به خصوص در L1

چرا hit time را کم کنیم چون ۹۹٪ ریکوئست‌های پردازنده داره hit می‌شه اگر hit time یک بلاک بیشتر شه عملکرد پردازنده دو برابر می‌شه به همین دلیل در L1 تا جایی که می‌تونیم کش را کوچک و ساده می‌سازیم به اضافه این که از تکنیک way prediction هم می‌تونیم استفاده کنیم

way preidiction یعنی ما در یک کش associative داریم و یک ریکوئست به کش می‌ده و کش می‌خواد دنبال اون آدرس در یک set بگرده ایندفعه set تمام بلاک‌ها را check کنه و tag را با tag های اونجا مقایسه کنه تا نتیجه بگیره را پردازنده می‌خواد ، way prediction می‌گوید شما اگر می‌دونید پردازنده رفته سراغ فلان set و در این set هم 4 تا بلاک وجود داره کش یک پیش بینی کنه که پردازنده کدوم یکی از این بلاک‌ها را می‌خواد چه جوری؟ حدس و گمان‌ها مبنی بر بدست آمده است که می‌تونیم در نظر بگیریم که بلاک‌ها مورد رجوع بودن ، آن را پیش بینی کنیم که پردازنده الان آن را می‌خواد ساده‌ترین way prediction اینه که اول بلاکی که آخرین بار access شده است را پیش بینی کنیم

Scanned by CamScanner

دوباره می‌خواد وقتی این پیش بینی را داشته باشیم قبل از این که نتیجه‌ی compare کردن tag را با آدرس بدهیم

Year:..... Month:..... Day:.....

یکی از بلوک‌ها را به عنوان بلوک predict شده می‌خوانیم و آماده نگه می‌داریم و

کار compare یک گام به جلو می‌رود با احتمال 80 درست و قطعاً در سیستم برای پردازنده و با احتمال کافی

اشتباه است و باید بلاک دیگر را بخوانیم

۲. bandwith کش را افزایش بدهیم

فارق از این که latency چند باشد می‌توانیم bandwidth را pipelined کردن کش خاصی

بدهیم

یک access کش عملکرد ۱۰ کلاک طول بکشد ولی برای این که برای هر ریکوئست ۱۰ کلاک صبر

نشد و هر جا بتواند چند ریکوئست را انجام دهد از pipeline استفاده کنیم

از این طرف می‌شه کش را multiband کرد یعنی کل فضای حافظه را تقسیم کنیم به چندین بانک حافظه که هر بانک

مستقل از بقیه‌ی بانک‌ها این قابلیت را داشته باشد که به ریکوئست‌ها سرویس دهد آن وقت می‌توانیم

به چند بانک کش access کنیم و چند بلاک را با هم بخوانیم (یکی خونه یکی دیگه بنویسیم و ...)

کار دیگه‌ای که می‌شه کرد از non-blocking cache استفاده کرد یعنی وقتی به یک کش

یک ریکوئست می‌دیم و این miss می‌شه باید یه بلوک را از حافظه بخونه و بیاره بنویسه تا بتونه

کارش را انجام بده که این کار چند ده کلاک طول می‌کشه کش در این بازه دیگه کار جدیدی برای cpu

انجام نمی‌ده non-blocking cpu به بلوکی که خواسته می‌رسه انجام و باید برای RAM بخونه

و کمی هم طول می‌کشه تا این کار انجام می‌شه اگر چند دیگه ای می‌خوای بنویسیم برات انجام بدهم

۳. miss penalty را کاهش بدیم یعنی هزینه‌ی زمانی که برای آوردن بلاک از حافظه به کش انجام می‌دهیم

تکنیک critical word first و merging write buffer راه‌های این روش است

۲. با نیتیم یک write buffer داریم که هر line آن می‌تونه معادل یک بلاک کش باشه زمانی که پردازنده

word به word کار دیتا را در write buffer می‌نویسه از این طرف write buffer هم

می‌تونه بلاک‌های کش این دیتاهایی که داره نوشته می‌شه مثلاً بسته در حافظه merging می‌شه این که

پردازنده وقتی داره یک دیتا می‌نویسه این دیتا را بلافاصله در حافظه نمی‌نویسیم و صبر کنیم شاید

Scanned by CamScanner

پر از زنده دیتا های دیگه هم داشته باشه که در یک بلاک بیفتن و صبر بکنیم یک دسته ای از دیتا را به سمت حافظه بفرستیم

Year:..... Month:..... Day:.....

۴) miss rate را کاهش بدیم بدون این که بخوایم کش یا بخت ازار را دست کاری کنیم، با استفاده از تکنیک های کامپایلر یعنی برنامه نویس یا کامپایلر اثر بدون بخت ازار چه جوری کار می کنه، ساختار کش چه جوری بلاک سایز چنده، سایز کش چند هست و ... می تواند بلاک ها را در حافظه و الگوی دسترسی به حافظه رو بدون این که خیلی برنامه تغییر کنه جوری عوض کنه که از این واحدی را زیادتر استفاده می شود.

۵) miss penalty یا miss rate را کم کنیم با استفاده از parallelization یا موازی که در سطوح مختلف می توانیم انجام دهیم چه بخت ازاری و چه بخش اثر ازاری با استفاده از روش prefetching. ما می توانیم با استفاده از پیش واکشی یا prefetching هم miss penalty رو کم کنیم هم miss rate رو کم کنیم

اسلاید ۶:

این شکل ۵ تا کش (۵ سایز مختلف کش) در نظر گرفته و محور y، Relative access time را نشان می دهد. این نمودار می خواد بگه وقتی سایز کش زیاد می شه access time کش چه جوری تغییر می کنه به اضافه این که برای هر سایز کش ۴ نمودار به ازای associativity های مختلف را نشان می دهد یعنی هم اثر سایز کش هم اثر associativity بر access time را نشان می دهد.

کش ۱۶K را در نظر بگیریم، این نمودار مشکی می شه direct maps بعدی 2-way و بعدی 4-way و بعدی هم 8-way. اینطوری ترسیم شدم access time چه جوری می رود بالا و می associativity می رود بالا. از direct تا 8-way، access time سه برابر شده

همین اتفاق در کش ۳۲ بیتی در نظر بگیریم direct تا 2-way access time برابر دارند

در ۶۴K اون کش direct، access time بیشتری از 2-way دارد، در ۱۲۸K و ۲۵۶ هم فاصله ها کمتر می شود

نتیجه ای که می گیریم، هر چه سایز کش را بزرگتر کنیم نفس سایز کش به associativity بیشتر می شه

اعداد های عجیبی هم در این نمودار می بینیم مثلا access time در 2-way در ۱۶K و ۳۲K تغییری نکرده یا

Scanned by CamScanner

در ۴۲k از 2-way ~~direct~~ کمتر است یا direct ، ۲۵۶k و 2-way و 4-way یکسان است

Year:..... Month:..... Day:.....

در ۱۶k و ۲۵۶k access time ، 8-way بهترین یا به اندازه است و در ۳۲k ، ۶۴k و ۱۲۸k در 8-way access time کمتری نسبت به ۱۶k دارد

* پس نتیجه می‌گیریم طبق این نمودار هرچه سایزش بیشتر می‌شه بعضی associativity در افزایش access time کمتر می‌شه

اسلاید ۷:

این نمودار energy per read نسبت به سایزش را نشان می‌دهد

هرچه associativity بالاتر می‌رود انرژی چندبرابر می‌شود. تقریباً روی 8-way access time روی ۶ تا ۱۹ است و به سبت associativity که بالاتر رفته انرژی هم بالاتر رفته. در ۱۲۸k و ۲۵۶ رفتارشون در direct و 2-way خیلی تغییری نکرده

* در این نمودار می‌بینیم بعضی associativity با افزایش سایز خیلی کمک نمی‌کنه ولی سایزی که بالا می‌ره انرژی به چون نسبت چندبرابر نشده

این نمودارها جزو معیارهای تصمیم‌گیری هستند

اسلاید ۸:

طریقه‌ای که در کش می‌خوایم برای این که access time را کاهش بدیم این است که access به کش و کش set associative ها access به تاش را parallel می‌کنن: access tag

در direct یک بلوک دیتا اگر باشه یک جای مشخص است اگر نباشه دیگه نیست پس وقتی داریم tag را compare می‌کنیم اون بلوک را هم بفرستان بیاش بخونیم

ولی در set associative می‌دونیم یکی از n تا بلاک یک set شده hit شده چون می‌دونیم چند تا tag تا کش تو شبکه compare را انجام دهند دیتا را بخونیم

Scanned by CamScanner

در cache parallel access (کش associative) دو نوع access داره یک نوع prallel

Year:..... Month:..... Day:.....

یکی serial access) در این صورت عمل می‌کند که اگر این شکل اسلایده

یک کش 4-way باشد و این خوشه‌ها با هم یک set باشند زمانی که ما از قسمت index شماره‌ی

set را پیدا می‌کنیم و می‌آییم به این ردیف از کش می‌رسیم (ردیف خوشه) در درون ست هستیم از ۱ تا ۴، tag

را کاهش می‌کنیم با tag های جدول‌ها و دیگری که hit و یک عامل آن را می‌خوهیم در prallel access

tag آدرس را با tag های جدول‌ها با چهار تا comparator همزمان compare می‌کنیم که باید valid

آنها را هم در compare کردن در نظر بگیریم، همزمان با خواندن tag و بیت valid و compare کردن،

Data های ۴ تا جدول همزمان با هم خوانده می‌شوند. تا زمانی که tag ها خوانده و compare می‌شوند

دیتاهای درون به آن مالتی پلکسر و کاندیدا از ۴ تا بیت داخل مالتی پلکسر وارد شدن درون روی Bus و خروجی

comparator ها با هم or می‌شوند که فهمیم hit شده یا miss شده

اسلاید 39

pipelined cache

* کش پایپ لاین اینجوری هست که access time ها کاهش نمی‌ده چه بسا بیشتر هم می‌کند ولی bandwidth

بالا می‌ره چون کش در هر کلاک یک ریکوئست می‌تونه بگیره به عنوان مثال در Pentium، Latency کش

ها 1 cycle بوده در Pentium 3 شده 2 clock و Pentium 4 شده 4 clock که خب از نظر

access time این به صرفه است ولی به این دلیل تغییر شده که pipeline داره و به جای pipeline در هر کلاک

می‌تونه یک پاسخ بده. البته وقتی pipeline می‌کنیم هزینه‌ی مربوط به mis-prediction ها branch

افزایش می‌دهیم. وقتی کش pipeline باشد در آن واحد چندین ریکوئست را داره هندل می‌کند و در یک لحظه

ممکنه 4 تا ریکوئست‌های کدام می‌خوام چون من منتظر می‌کردم branch آیا داره این در می‌ره

ولی اون در می‌رفت این مشکلی است که کش pipline ایجاد می‌کند.

* وقتی کش pipeline باشد این که بتوانیم associativity را بالا ببریم بهتر می‌شود چون کمتر شدن افزایش

Latency کش هستیم

Scanned by CamScanner

اسلاید 10

Year:..... Month:..... Day:.....

multibanked cache

کش می‌تواند تقسیم شده به چندین بانک حافظه که آنجا می‌توان هم‌زمان access شن. بنابراین می‌توانیم بلاک‌های حافظه را در این بانک‌های حافظه پخش کنیم و وقتی عدل می‌زنیم که مثلاً دو تا بلاک پشت سر هم access شن این دو تا بلاک را در دو تا بانک مجزا از هم بزاریم که اگر دو تا ریکوئست پشت سر هم به این دو بلاک شد یکی معطل اون یکی نشه

اسلاید 11

*nonblocking cache: در حالی هست که کش miss می‌شه ولی نخوایم پردازنده را معطل نگه داریم. پردازنده یک ریکوئست داده و دیتاش مورد نیازش را نداریم اما پردازنده‌ای که این قابلیت را داره که به صورت multi thread برنامه‌ها را اجرا کند + اجرای خارج از ترتیب داشته باشد این پردازنده اگر یک ریکوئست اجابت نشد آن را برای کار و کارهای بعدی را انجام بده و ما هم باید کش را طوری درست کنیم که به این نیاز پردازنده پاسخ بده

nonblocking حالت hit under miss یعنی اگر یک miss رخ داد کش ما تا اون miss دیتاش می‌رسد به پردازنده می‌ره ریکوئست بعدی را نده اگر ریکوئست بعدی hit بود من بلاک را بجت میدم و بخاطر قبلی معطل نمی‌کنم حالا اگر ریکوئست دیگه اومد و اون هم miss شد کش به پردازنده می‌گه اون قبلی را هنوز نیاوردم باید صبر کنی قبلی را بیارم بجت بدم این هم که الان miss شده برات بیارم بعد بقیه کارهات را راه می‌اندازم پردازنده می‌گه دو تا miss دارم ولی چند تا دیگه دارم که hit می‌شه آیا بازم بهم برگشت می‌تونی بدی بهم؟ کش می‌گه اگر ساختار hit under multiple miss باشه می‌تونم این کار را انجام بدم یعنی من که به ریکوئست‌های بعدی دارم جواب میدم یک بافری دارم که ریکوئست قبلیت که miss شده نداشتم بری بافر و اون داره کار خودش را انجام میده و من هم به کارهای دیگه‌ات می‌رسم و در صورتی می‌تونم زمانی یکی دیگه miss می‌شه بقیه دستورات را انجام بدم یک بافر دیگه داشته باشم که این که miss شده را بزارم اونجا

- کش L2 باید بتواند چنین ساختاری داشته باشد بخاطر این که L2 تا L3 و main memory راه درازی در پیش دارد

Scanned by CamScanner

- اما L1 شاید نیازی به این ساختار non-blocking نداشته باشد چون non-blocking

هزینه داره و کنترلر کش را پیچیده‌تر می‌کنه و کارایی کش را سخت‌تر می‌کنه

Year: Month: Day:

به این دلیل که L1 کش نیاز ندارم که L1 نیاز دارم که miss شبه پردازنده می‌تواند کارهای کشه‌ای انجام دهد و

مشکل کش L1 نباشد ، L1 اگر miss شبه ۴ ، ۵ کلاک بلد دیتا به دست پردازنده می‌تونه برسه از L2 چون

فاصله‌ی L1 تا L2 زیاد نیست ، پردازنده هم که pipeline داره و اگر یکی miss شبه به دستورات می‌تونن

برن جلو اما اگر در L2 ، miss رخ داد یعنی L1 ، miss شده ، L2 هم miss شده و

پردازنده باید تا ۱۰ ، ۲۰ کلاک صبر کنه که دیتا کش برسه اینطوری است که L2 باید بتواند ریکوئست‌های بعدی را

جواب بده تا پردازنده کمتر معطل بشه

حالا سؤال اینه در حالت hit under multiple این multiple چند باشد؟ 1 ، 2 ، 4 ، 8 ، 16 ....

miss را تحمل کنیم؟

تصمیم با ماست که چند برابر از این بارها بدهیم که تا چند تا miss اجازه شده را تو

تحمل کنه و همچنان تو به پردازنده سرویس بده

این بار benchmark ، spec cpu ، integer و Floating point را نشان می‌ده که

معتبرترین benchmark است که در این زمینه استفاده می‌شه . در این benchmark ها پردازنده‌ای

تست کردن از منظر cache access latency در حالت‌های مختلف . یک جا cache blocking

باشه یعنی این که اگر miss رخ داد کاری نکنید تا کار این miss تموم شه . اگر در این حالت cache access

معلق باشد در حالت hit-under-1miss ، hit-under-2mis ، hit-under-64miss بررسی کردن

چیزی که در این شکل دیده می‌شود این است که تقریباً روی همه‌ی benchmark ها همین که بتونیم یک miss را

تحمل کنیم کفایت می‌کنه و فاصله بین 1miss و 2-miss و 64 miss خیلی ناچیز است

این کشه مراحل کش خیلی امیدوار نباش که همین جور اگر قابلیت به کش اضافه بکنی بهبودی در کارایی کش ببینی و

ممکنه هزینه زیادی داشته باشه و به صرفه نباشه

همین اتفاق در زمان associativity هم می‌افته از یه جایی به بعد نباید دیگه بهبودی ببینیم

Scanned by CamScanner

اسلاید ۱۲:

Year:..... Month:..... Day:.....

Critical word first: این تکنیک برای کاهش penalty miss است. زمانی که درگیر miss رخ میده این miss خاطر رنگو شت پردازنده به یکی از اون بلاک هایی است که درگیر نبوده این تکنیک می گه وقتی می ری سراغ حافظه و می خواد اون بلاک را بیاره به جای این که از اول بلاک شروع کنی و تا آخرش بگی بعد را بده اول اون دیتایی که وسط اون بلاک می خوای رو به حافظه بگو بفرست بده و اون دیتا رو به دست پردازنده برسون ما بقی اون بلاک را پشت سر اون دیتا بیار (یعنی اول دیتای مورد نیاز را بیار بعد کل بلاک را بیار)

- Early restart می گه به جای این که از وسط یک بلاک بود دیتا را برداری بدی به یک پردازنده بلاک را طبق حالت عادی بخون و دونه دونه دیتا را از اول تا آخر بخون به پردازنده مثلاً دیتای وسط را خواسته، دیتای اول را کش می ذاری، بعد دومی را درکش می ذاری، سومی را درکش می ذاری و همینطور که پردازنده می خواد Early restart می گه آقای کش صبر نکن کل بلاک از حافظه به دست برسه کنار اون بلاک خودت بنویسی و بعد یکی از وسط برداری به پردازنده بدی. اون دیتاهایی که دونه دونه به دست می رسن هر وقت رسید به اون که پردازنده می خواست بخون لحظه بعش بده تا بقیه که می رسن اونها را کنار هم بزار

* - این روش ها به شرطی می تونن تاثیر گذار باشن که بلاک بیاز ما از حوزه بزرگتر باشه چون زمان بردن طولانی تر است

اسلاید ۱۳:

تا اینجا هر کاری کنیم سخت افزار انجام میده حالا باید بیایم نرم افزاری کامپایلر برنامه نویس چه کاری برای بهبود کش می کنه

یکی از کارهایی که در سطح نرم افزاری بهبود performance می شه کرد تکنیک روش loop interchange است که این روش در حالتی که loop های تو در تو داریم استفاده می شه. یعنی که برنامه نویس loop های تو در تو می نویسه برایش مهم نیست که این آرایه ها چه جوری کنار هم قرار می گیرن و برایش مرتبی نیست که این ماتریس را سطری بیایش کند یا ستونی در حالی که افرادی مثل ما می دونن که وقتی ماتریسی را به صورت عمودی بیایش می کنیم درایه هایش با فاصله به اندازه ی طول سطر از هم قرار گرفتند و وقتی یک درایه از یک ماتریس را می خوریم یک بلاک از حافظه خونده می شه می ره توی کش و این بلاک یعنی اون درایه و درایه های مجاورش در اون ردیف

Scanned by CamScanner

وقتی به صورت سنتی می‌رسیم خط بعدی و درایه‌ی بعدی را می‌خوانیم به ازای اون تک درایه یک بلاک کش می‌شه و n تا درایه از اون ردیف خوانده می‌شه. داریم ماتریس را از عرض می‌رسیم سطر به سطر و از هر ردیف فقط یک درایه

Year:..... Month:..... Day:.....

را می‌خوانیم ولی بخت از این بیچاره به اصطلاح این که وقتی یک درایه access شد درایه‌های مجاورش که همون ردیف access بشن و اره همین جور مکرر دیتا از حافظه می‌خوان به کش می‌برن که همه بلا استفاده هستند

پس برنامه‌نویس وقتی این قضیه را می‌دانست access به حافظه و دیتا را جوری تنظیم کن که مطابقت داشته باشه با چیدمان دیتا در حافظه و نحوه‌ی خوندن و انتقال دیتا از حافظه به کش

- تکنیک Blocking می‌گه که وقتی ما می‌خوایم سطر و ستون یک ماتریس را بخوانیم به جای این که در هر ردیف یا ستون از سر تا ته برویم به صورت بلاک بلاک بخوانیم و پیمایش به صورت بلوکی باشه، یعنی این جوری نباشه که یک ردیف را تا آخر بخوانیم تا برسیم به ردیف بعدی، ۴ تا از این ردیف بخون بعد دوباره برگردی ردیف اول ۴ تای بعدی را بخون و الی آخر... بسته به این که سایز دیتا چقدر باشه و الگوی دسترسی به چه صورتی باشه

اسلاید ۱۴:

مثالی برای توضیحات بالا

تکه کد مربوط به این اسلاید برای ضرب ماتریس‌هاست به این صورت عمل می‌کنه که y_i و k را در z_k و x ضرب می‌کنیم می‌دانیم در x اگر جای For مربوط به i و k را عوض کنیم کارایی برنامه ما سریع‌تر می‌شه و cache miss ها کاهش پیدا می‌کنه (این همون تکنیک Loop interchange است)

اسلاید ۱۵:

مثال Blocking که می‌گه می‌گه با For ها $y^{\sim}$ را در $z^{\sim}$ ضرب می‌کنه در کلاس داره با این جور ضرب کردن شاید cache miss ها کم شه، تعداد access به حافظه کم شه، شاید پردازنده بتونه خیلی خواندن یا گرفتن دیتا به کش شه و سرعت بالا بره

اسلاید ۱۶:

راهکار دیگه بحث prefetching یا پیش‌واکشی است که هم سخت‌افزاری است هم نرم‌افزاری پیش‌واکشی یعنی ما تا الان وقتی miss در کش رخ می‌داد می‌رفتیم سراغ حافظه و اون بلاک را می‌آوردیم، پیش‌واکشی می‌شه زمانی که پردازنده داره کار خودش را می‌کنه و کش خوشحاله که داره hit می‌شه، اگر حافظه آزاده و اون

Scanned by CamScanner

نیست خبری پس رکاری نداری از اون در برو جیند تا بلاک بیار یعنی قبل از این که miss رخ بده یک حدس بزن که

این cpu با این رفتار در آینده ی نزدیک چی می خواد که الان نداری برو همون را

بردار از حافظه بیار

ساده ترین حالت prefetching این است که زمانی که یک بلوک miss می شود همراه با اون بلوکی که miss شده

بلوک بعدی یا قبلی را بیار (cpu یک بلا بلوک خواسته من که تا از بار فتم بلوک بغلی را میارم احتمالا cpu

اون را می خواد)

نمودار این اسلاید با همین تکنیک ساده می که در برنامه های spec int 2000 و spectfp2000 چه

کارایی می توسم ببینیم

اسلاید ۱۷

حالت نرم افزاری پیش واکشی اینه که کامپایلر با علم به این که بی درونه وقتی داره این مسیر را می ره در چند خط

جلوتر به این دیتا ها نیاز داره، یکسری دستورات عمدتا داخل دستورات برنامه براه که اون دستورات ما نخت افزار

را وادار می کنه که دیتا هایی که در آینده نزدیک بهشون احتیاج داره بیاره در کش براه

این prefetching ها حالت Faulting و non-falting دارن می تونند خواسته باشند که منجر به

exception در سیستم عامل شه

memory managment در سطح سیستم عامل پیشرفته است که ممکن است پردازنده چیزی بخواهد و در حافظه

نباشد و page fault رخ بده

این پیش واکشی در سطح رجیستر هم می تونه انجام بشه در رجیستر ها توسط کامپایلر handle می شه، دیتا ها هم

باید در رجیستر بیرن تا پردازش بشن در این prefetching کامپایلر یکسری دستورات می ذاره که پردازنده را

را وادار می کنه که در لابلای کارهاش یکسری دیتا ها را رو براه داخل رجیستر هایی که در آینده نزدیک

احتمال داره استفاده بشن

پیش واکشی در سطح کامپایلر خوب است که ترکیب شود با تکنیک loop unroling

loop unroling یعنی باز کردن یک loop یعنی یک loop که با ۴ خط کار می کنه و Itretion آن صد بار است را

باز کنیم و ۴ خط را بکنیم ۲۰ خط Itretion را به جای ۱۰۰ بار بکنیم ۲۰ بار . [illegible]

برنامه نویس باید حواسش به این مورد باشه که loop ها را باز کنه تا Itretion کمتر شه و هزینه را کمتر کند

Scanned by CamScanner

«جلسه‌ی چهارم»

Year:..... Month:..... Day:.....

ویدیو اول (Lec4-Aca)

اسلاید ۳:

- prefetching چیه؟ چرا خوب است و چه کاری می‌کند؟

prefetching یعنی آوردن چیزی قبل از این که به آن نیاز داشته باشیم، به امید این که در آینده‌ی نزدیک نیاز شود

- در مورد prefetching ۴ سؤال وجود دارد:

چه چیزی را prefetch کنیم، کی بیاریم، کجا جا بدهیم، و prefetching را به کی بسپاریم

- برای prefetching هم روش نخست افزاری داریم هم نرم افزاری. یعنی می‌توانیم الگوریتم prefetching را به نرم افزار بسپاریم یا به سخت افزار

- معیارهای برای خوب بودن prefetching:

timeliness ، accuracy ، coverage

تعداد miss های cache miss که با prefetching miss نشدن

Coverage = Cache misses eliminated by perfetching / Total cache missess

تعداد cache miss ها

prefetching دقت

prefetch Accuracy = cache misses eliminated prefetching / (useless cache prefetches) + (cache misses eliminted by prefetch)

از این چیزهایی که prefetcher آورده چند درصدش به درد خورده چند درصدش به درد نخورده

(تعداد prefetching هایی که miss نداشتند) (تعداد prefetching هایی که miss نداشتند)

یعنی دیتایی را بیاریم در cache و access نشه و استفاده شه

timeliness یعنی کاری را سر وقت انجام دادن. سر وقت انجام دادن prefetching یعنی اگر می‌دانیم یا حدس می‌زنیم بلاکی از حافظه به زودی را کش رفتوشت داده خواهد شد و miss خواهد شد آن بلوک را سر وقت برداریم بیاریم در کش

نه خیلی زود نه خیلی دیر

Scanned by CamScanner

مورد دیگه ای که باید حواسمون بهش باشه cache pollution و Band width

Year:..... Month:..... Day:.....

prefetching یعنی این که ما Bus بین cache و main memory که

ظرفیت کم و دری دارد و Band width مشخصی دارد و قرار است استفاده ی مفید از آن شود باید به ظاهر

کارهای بیهوده prefetching حذف کنیم

prefetching اگر دقتش پایان باشه به درد می خوره و ترجیح می دیم از آن استفاده بکنیم چون Bus

را اشغال می کند و کسانی که با Bus کار می کنند و بدرد بخور دارند را داره علل می کنه

* cache pollution یا آلوده کردن cache یعنی چیزهایی را بیاریم داخل cache که بدرد نمی خوره و به ازای

آن چیزهای مفید را بریزیم دور

* cache Trashing یعنی بلوک های کش هی بیان و بروند هم دیگه را بیرون بیاندازند و این دعوا و شلوغی های

که در کش اتفاق می افتد اصطلاح بیهوده ای است اگرچه بلوکی که می آید بدرد بخوره ولی قبل از این که Access

شده دور ریخته می شه و بخواه دوباره برگرده

اسلاید ۴:

ایده کلی prefetching: دیتا را قبل از این که درخواست بشه و نیاز باشد fetch کنیم (هم بلوک های

دیتا هم instruction می توانی fetch بکنی)

چرا درس داریم prefetching انجام بدیم؟

چون memory latency و memory خیلی زیاد برای ما دوس نداریم latency حافظه را نیستم

اگر بتوانیم بلوکی که مورد نیاز است است را قبل از این که درخواست بدن بیاریم در کش زمان را خریدیم

و تعلق ها را از بین می بریم. اگر دقت بالا باشه به موقع بتوانیم این کار را انجام بدیم می توانیم کاملا

latency هم prefetcher نیستم. eliminate latency کنیم که جلو access به حافظه داریم. در واقع

داره این latency زمانی است که Bus آزاد است و memory بیکار است و دیر بازده هم

مشغول است به همین دلیل prefetching باعث می شود که ما بتوانیم compulsory miss

را بیرون از بین ببریم (نه این که کامل از بین ببریم بلکه درصد آن را کم کنیم)

Scanned by CamScanner

آیا می‌شه Conflict و capacity را با prefetching کمتر کرد؟ بله

Year: Month: Day:

prefetching لازمه‌ش این است که ما بتونیم predict کنیم چه آدرسی در آینده می‌خواهد رفرنس شود و این کار در صورتی که رفتار برنامه‌ی ما predict باشد می‌تواند با دقت بالایی انجام شود. Predict بودن برنامه یعنی این که رفتار آینده‌ی برنامه با رفتار گذشته‌اش تطابق داشته باشد

آیا با استفاده از روش prefetching تعداد access های پردازنده به حافظه کاهش پیدا می‌کند یا خیر؟

با استفاده از prefetching علاوه بر این زیادی از miss ها رو می‌تونیم از بین ببریم و miss rate کاهش بدیم

اسلاید ۵:

آیا prefetching اشتباه باشد correctness برنامه‌ی ما سؤال می‌رود؟

* خیر چون خود cache بود و شود در correctness و Functionality ندارد بنابراین اگر misprediction رخ بدهد در prefetching ما هیچ نیازی به state recovery در پردازنده‌ی خودمان مجبور نیستیم همه چیز را برگردونیم به حالت قبل

* و این برخلاف پیش بینی‌های دیگه‌ای است که در پردازنده داریم. ما در پردازنده branch prediction می‌تونیم پیش بینی کنیم و اگر درست از آب درنیومد بیخیال بشیم و بریم اتفاقی نیفتاده بلکه باید برگردیم به اول و مسیر درست را برویم و value prediction داریم

اسلاید ۶:

طبیعی است از روش prefetching با سایر بلاک‌ها کش رفت و آمد انجام می‌شه در prefetcher

- prefetching معمولاً روی کش اتفاق می‌افتد و با دانه‌بندی کش بلاک انجام می‌شه

- prefetching هم می‌تواند miss rate را کم کند هم miss latency را کم کند

prefetching می‌تواند باعث شود هم تعداد miss رخ داده کمتر شود هم latency آن کمتر شود این دو حالتی است که prefetcher ←

Scanned by CamScanner

یک چیزی را پیش بینی کرده که بیاره و زحمت تا نصف راه را رسیده و در حال آوردن دیتا است که پردازنده آن را درخواست

Year:..... Month:..... Day:.....

میده اینجا miss رخ میده چون کش آن اندازه اطلاعات prefetch نمی‌ره

زحمت و بلاک را اداره میاره اگر یک ذره زودتر آورده بود miss رخ نمیداد اما الان هم یک بخشی از کار انجام شده

و miss latency کاهش پیدا می‌کند و مقدار پردازنده کم می‌شه

- prefetching می‌تواند توسط

سخت افزار انجام شد بدون دخالت کامپایلر و نرم افزار

کاملا توسط کامپایلر انجام شد به صورت نرم افزاری

یا به صورت نرم افزاری توسط programer انجام شود

اسلاید ۸۷

چه زمانی prefetch شود؟

الگوی prefetch کی انجام شود؟

کافی داریم چیزی که اومده؟

چه چیزی انجام بشه؟

* به نحوه‌ی انجامش می‌گه software ، hardware ، execution-based

یا ترکیبی از اینها باشد

کامپایلر یا programer انجام دهد

یک قطعه کد داشته باشیم

این قطعه کد یا بخشی از نرم افزار

باشد یا بخشی از یک کد سخت افزاری

باشد و در یک زمان‌های خیلی خاصی

یک چیزی احتمال داشته که به طریق اون

یک چیزی از حافظه به کش آورده شود

اسلاید ۸۸

چه زمانی prefetch شود؟

اگر ما قبلا بدونیم منابع resource در دارایم یعنی Bandwidth حافظه را بهش اشاره کردیم

کش را شلوغ نکردیم دکارهایی که بهش احتیاج داشته برنامه به تعویق انداختیم

Scanned by CamScanner

فضای حافظه کش را هدر دادیم. ما کشی داشتیم که کار می‌کرد و بلوک نگه می‌داشت اما ان گفتیم یه چیزی را بندازیم دور که

Year:..... Month:..... Day:.....

ما داریم یک چیزی را prefetch می‌کنیم اگر بدرد نخور باشه عملاً دیتای مفید

آن را دور انداختیم و این باعث می‌شه که miss rate ما زیادتر هم بشه

بخاطر prefetching اشتباه طبق انرژی هدر رفته، هم در کش هم در Bus هم در main memory

که این منابع می‌تونست استفاده بهتر از شون بشه و این منابع را می‌توستیم برای demand request ها

یا demand این منابع را هدر دهیم یعنی هر وقت که لازم شد از این منابع استفاده می‌کردیم یا بیشتر هزینه می‌کردیم

تا prefetching دقیق‌تری می‌داشتیم

سوال مهم: آیا وجود prefetcher و prefetch کردن دیتاها و آوردنشون قبل از موعد آیا در مصرف

انرژی سیستم تاثیری دارد؟ تاثیر هدر مثبت دارد یا منفی؟

- پس تمام تلاشمون را باید بکنیم آن چیزی که قرار است prefetch کنیم یک prefetching بدرد بخور

باشد که بحث Accurate مطرح می‌شود

prefetch Accuracy = used prefetches / sent prefetches

تعداد کل prefetch هایی که انجام دادیم / تعداد prefetching هایی که بدرد بخور بوده

مسئله اینه که از کجا بدونیم چه چیزی را بیاریم یعنی پیش بینون بر مبنای چه باشد؟

هر وقت بخوام چیزی را پیش بینی کنم یعنی همان گذشته است

راه دیگه اینه که از کامپایلر بپرسیم که آیا دستورهای به دیتاها چطور است و کامپایلر یک سری راهنمایی ها

بکند یا خود برنامه نویس بتواند با آنالیزهایی که روی برنامه انجام میده یک پیش بینی‌هایی بکند

- پس ما یک prefetching algorithm داریم که تصمیم می‌گیرد چه چیزی بیاورد.

prefetching algorithm چیزی شبیه cache replacement algorithm است که در اونجا تصمیم می‌گیریم از کش

چیزهایی که در کش داریم چه کسی را بیاندازیم بیرون در prefetching algorithm می‌گیریم از این جاهایی که در

کش نیستند چه کسی را بیاریم در کش

Scanned by CamScanner

اسلاید ۹:

Year:..... Month:..... Day:.....

بعد از این که تصمیم گرفتیم چه چیزی بیاوریم باید ببینیم آن چیز را چه زمانی بیاوریم و

که آن را initiated کنیم روی حافظه؟

برای این، این مسئله خیلی مهمه که اگر چیزی را که می‌آوردیم خیلی زود بیاوریم یعنی فاصله‌ی بین آوردنش تا اولین ریکوئستی که بهش میشه زیاد باشه حتی اگر چیز درست آورده باشیم ممکن است قبل از این که استفاده بشه بیاندازنش بیرون. این زود آوردن اتفاق یک prediction غلط آوردیم چون الان به یک چیز دیگه احتیاج داشتیم

خیلی دیر هم نباید بیاریم یعنی فاصله زمانی بین اقدام کردن برای بلاک تا آوردنش اگر خیلی کم باشه ممکنه قبل از این که بلاک به مقصد برسه و درخواست برای اون ریکوئست بشه و ما بتونیم کاملاً latency را پوشش بدیم و miss رخ میده ولی جلب miss latency کاهش پیدا می‌کنه ولی کاملاً از بین نمی‌ره

حالت ایده‌آل این است که بلاک را بیاریم قبل از این که ریکوئست بشه و Access بشه قبل از این که اجرا بشه

پس این بحث timeliness که قبلاً داشتیم این که prefetcher ما چقدر کارش را به موقع انجام میده و چه قدر خوب است به همین مسئله که گفتم قرار بیاره دست نگه داره

اگر سیستم ما بتونه اگر سیستم خیلی خوبی باشه یعنی هم coverage بالایی باشه هم accuracy بالایی داشته باشه timeliness بالایی داشته باشه اما به ازای این کاری که قرار انجام میده ممکن است که محاسبات خیلی زیادی لازم داشته باشد مدار خیلی پیچیده‌ای داشته باشد و اطلاعات خیلی زیادی لازم داشته باشد که این هم ممکنه بد بخواد مثلاً ما ترجیح بدیم prefetcher ما از ۱۰ تا ۲۰ تا ۳۰ تا بیشتر خطا کند به همین مقدار کمتر به موقع بیاره ولی این همه هزینه به سیستم ما تحمیل نکند

اسلاید ۱۰:

حالا باید ببینیم کجا بذاریمش:

- درکش بذاریم
- یا کنار کش یک بافر بذاریم که prefetch buffer باشه و آن چیزی که prefetch کردیم در خود

Scanned by CamScanner

کش نداریم و در این بافر نداریم چون نمیدونیم این چیزی که برد بخور هست یا نه در بافر می‌ذاریم که کش کار خودش را انجام بده و همه اصم کش بسیم

Year:..... Month:..... Day:.....

اگر بدون کش بذاریم:

چیزی خیلی ساده است و مثل cache miss است و وقتی موقع cache miss میاریم یه جایی می‌ذاشتیم این هم میاریم اونجا می‌ذاریم و یکی هم می‌اندازیم بیرون که این جدیده قراره بیره در واقع prefetching مثل cache miss عمل می‌کنه و این خوبش هست یعنی نیاز به بافر نداریم و مدار هم خیلی ساده است

عیب این روش اینه که prefetcher ها، هر بلاک میاره براون و کش هم که خالی بلاک توش باید بلاک هاشو بیاندازه بیرون که این چیزایی که prefetcher آورده داخل خودش نداره و ممکنه یکسری از این ها بدرد بخور نباشن و کش میای useful را دور بریزه و مسئله cache pollution پیش بیاد

اگر در بافر جدا بذاریم:

خوبیش اینه که چیزهایی که در کش داریم مجاور prefetching آنها در معرض خراب شدن و cache pollution نخواهیم داشت

عیب این روش اینه که سیستم پیچیده‌تر می‌شه و طراحی کمی سخت‌تر می‌شه و باید به یکسری سؤالات پاسخ بدیم:

- اول prefetch را کجا بذاریم buffer
- prefetch buffer به access به کش. یا وقتی پردازنده چیزی از حافظه می‌خواد اول بریم سراغ کش بسیم اگر در اونجا نبود بعد بریم ببینیم در بافر هست یا نه همزمان با access به کش بافر را هم access کنیم تا وقت آنها در مصالحه بین performance و انرژی است. اگر موازی انجام بدیم همیشه داریم یکی از آنها را بیهوده access می‌کنیم و انرژی هدر رفته. اگر سریال access کنیم اول مواقعی که در کش نیست دیگر می‌رویم سراغ بافر یعنی تأخیری داریم

Scanned by CamScanner

• دیتایی که prefetch کردم و در prefetch buffer هست می‌برادم توی کش. برای این نیاز هست در کش بزارم که بافر هم خیلی زودی پر نشه و سایز بافر از سایز کش خیلی کوچکتر هست

Year:..... Month:..... Day:.....

آن چیزی که در بافر نداشتیم و برداشته آن را رکوئست کرده یعنی چیزی که داریم بعید بوده پس hit شده در بافر و بره در کش و در جای درست قرار گیره

حالا اگر یکسری بلاک داریم در بافر و بافر پر شده حالا اگر یک بلاک دیگه بخواد که بیاد بخوره هست یا اونم میره بیاره یا نه و یکی از اون چیزهایی که در بافر آوردیم و بعد دیگه خواه و حضور hit نشده دور بیاندازیم تا جا برای اون بلاک جدید را جایگزین کنیم. یا بهتر نیست اون بلاکی که در کش هست که خیلی بی‌مصرف هست و در LRU ته صف هستند بندازیم دور و آن چیزی که در بافر هست و حدس می‌زنیم دسترسی بشه بزاریم جای اون؟ این مسئله‌ای است که باید در موردش تصمیم گیری شه

• این سایز بافر چقدر باشه این سایز می‌تونه تابع دقت prefetcher هم باشه که هر چه دقیق‌تر باشه و timing بهتر باشه نیازها رو به بافر بزرگ کم کند

• این prefetch buffer با coherent شه یعنی سازگار با چیزهای دیگه

- خیلی از پردازنده‌ها از prefetcher استفاده می‌کنند و عده‌ی آنها چیزی که میاران در کش می‌ندازند و یک بافر اضافه کنار کش نمی‌ذارن

اسلاید ۵۱

درعین این که دیتا را کجا بزاریم یک بحث دیگه این است که prefetching را روی کدوم کش انجام بدیم و روی کدوم کش سوار شود

آن چیزی که prefetch شده است از حافظه در بیاریم، یا از حافظه بیاریم در L2 بزاریم یا از L2 بیاریم در L1 بزاریم. هر کدوم از اینها معایب و مزایای خودشون را دارن مثلا یک prefetch بیشتر نداشته باشیم و اون هم من main memory و L2 باشد

- یک نکته دیگه اینه که دیتایی که می‌آوریم و از حافظه و در کش می‌خوایم بزاریم (دیتایی که prefetch شده) جاش از نظر آدرس دهی جا مشخص است. بلاک‌هایی که miss می‌شد و می‌گفتیم کجا باید بروند این هم همون جا می‌رود بلاکی که prefetch شده دو حوزه میشه جش نگاه کرد ۱) شبیه بلاکی که

Scanned by CamScanner

miss شده و تازه وارد هست میاریمش میذاریم جای بلاکی که از همه پیرتره و خودش که تازه اومده همیشه جوان ترین بلاک در آن set (یعنی بلاکی که کمترین شانس بیرون انداخته شدن را داشته باشد) آیا این کار درست است که بلاکی که prefetch شده با بلاکی که برداشته از حافظه خواسته نه بلاک بهتر بوده بیرون بفرستیم احتیاج داره باید به یک چیزی نگاه کنیم؟ آیا عادلانه است که هر دو یک جور باشن و وقتی میان اول هفت تا از بیرون؟ بنظر ما این شاید تصمیم درستی نباشد باشه وارد بلاکی که prefetch باشه را اونقدر تحویل بگیریم که بلاکی که undemand شده را تحویل نمی‌گیریم. اونی که undemand اومده واقعاً لازم داشتیم که آوردیم و اونی که شانسی اومده ممکنه اصلاً اشتباه باشه شاید بهتر باشه به جایی به دیتا prefetch شده را در MRU position بذاریم بهتر باشه بذاریم در دورترین جای ممکن بذاریم پیرترین بلاک که خب این دو تا سیاست یکیش از این ور بوم افتاده یکیش از اون ور بوم و بین اینجا این می‌شه که اونی که prefetch شده را بیاریم وسط بشینه، اگر پیرترین باشه خب سریع می‌اندازیمش بیرون و کارهایی که ما کردیم برای prefetch هدر می‌ره و اگر هم که جوان ترین بذاریم تقریباً می‌اندازیمش بیرون و خودش ممکنه اصلاً access نشه

پس replacement policy ما شاید لازم باشد که یک تعادلی قائل شد بین اونایی که undemand شدن را اونایی که prefetch شده اند و شاید بهتر باشه اونی که prefetch شده در LRU position یا وسط بذاریم

اسلاید ۱۲:

اون hardware که کار prefetching را انجام میده کجا گذاشته شده؟

این prefetcher جایی گذاشته شده یکسری اطلاعات را می‌بینه یکسری را نمی‌بینه. prefetcher ها جنبه‌ای اطلاعاتی که رخ میده و history که از اونها داره باید تصمیم بگیره. این که چه اطلاعاتی را بتونه رصد کنه می‌تونه در تصمیم گیریش اثر بذاره

چیزهایی که می‌تونه ببینه:

- ممکنه جایی باشه که تمام hit ها و miss ها را ببینه
- ممکنه جایی باشه که miss ها را فقط ببینه
- ممکنه جایی باشه که miss ها ← prefetcher اطلاعات را بین ها و main memory گذاشتیم و اطلاعات بین ها و cache را نمی‌تونه ببینه

Scanned by CamScanner

اگر فقط miss ها را بتواند ببیند یعنی ما prefetcher را بدون یاد نداشتیم

Year:..... Month:..... Day:.....

- هر چه که pattern access بیشتر و یا کامل تری بینیم می توانیم انتظار داشته باشیم

که accuracy و coverage بیشتری داشته باشیم

اطلاعات زیاد داشتن یا رصد این است که پردازش بیشتری انجام بدهیم و هزینه ی بیشتری بکنیم و حافظه ی بیشتری مصرف کنیم و

کلاً حدس و گمان هایی که می خواهیم بزنیم خرج بیشتری روی دستمون بذاره

اسلاید ۱۳:

* چگونه prefetching را انجام دهیم؟

- prefetching را می توان به صورت نرم افزاری باشد

در این حالت دستورات prefetching را در کنار دستور خواندن و نوشتن و if و else داریم

این حالت با استفاده از این دستورات فقط وقتی به درد می خورد که ما بتوانیم آینده را ببینیم یا حالتی

باشد که برنامه access pattern منظم داشته باشد و قابل پیش بینی باشد

access

regular access pattern در مقابل pattern ✓ یعنی یک ماتریس است

- حالت سخت افزاری prefetching

سخت افزار همه ی کارها را انجام می دهد. یک حافظه یک کنترلر + حافظه ای که داده ها را نگه می دارد

چه اتفاقاتی بین کش ها می افتد را خود سخت افزار در مورد ذخیره می کند و بر مبنای آنها Run time

تصمیم می گیرد که چه کاری کند

- حالت Execution - base

در این حالت یک thread که کار prefetching را برای ما انجام دهد اجرا می شود که این thread

می تواند توسط برنامه نویس به وسیله یک دستور call شود یا توسط یک hardware فعال شود

اسلاید ۱۴:

* hardware prefetching:

در این روش یک سخت افزار خاص منظوره که observe می کند تمام تراکنش های load و store که

روی حافظه انجام می شود و بعد بر مبنای آن بر اساس آنچه در حافظه اتفاق افتاده است تصمیم گیری می کند

+ خوبی این کار سخت افزاری این است که این سخت افزار می تواند tuned شود بر اساس آن سیستم و سخت افزاری

که داریم یعنی بر اساس آن سایز کش و block size، سیاست جایگزینی که در کش وجود دارد...

اون prefetcher هم، چون کسی که این چیزها را طراحی کرده است طراحی می کند بنابراین ۵۸

در آن پردازنده همه چیز را بهم می‌خواند. ولی اگر توسط برنامه نویس به صورت نرم افزاری انجام شود

Year:..... Month:..... Day:.....

برنامه نویس از وضعیت کش و جزئیات سخت افزار اطلاعی ندارد

+ مزیت دیگر روش سخت افزاری این است که برای prefetch کردن مجبور نیستیم دستور را اجرا کنیم

- عیب این روش اینه که سخت افزار پیچیده می‌شه و المان و مدارهای پیچیده‌ای داره و علاوه بر افزایش مصرف انرژی باید مانیتور کند و حدس بزند

بنابراین روش سخت افزاری جایی مناسب است که قابل پیش بینی باشد

- روش‌های سخت افزاری کار برنامه نویس را سخت می‌کنند

اسلاید ۱۵:

* یکی از الگوریتم های prefetching، prefetchers next line است

ساده ترین روش prefetching سخت افزاری است، این روش می‌گه هر وقت بلاکی در کش miss شد در سیستم سراغ حافظه رفتن آن بلاک را از حافظه به کش آوردیم n تا بلاک بعد از آن بلاک را هم prefetch کنیم و از حافظه به کش بیاوریم (این n می‌تونه از 1 ... باشه)

روش خیلی ساده ای است prefetchers هیچ کاری انجام نمیده و هیچ history نیاز نیست نه دارد فقط هر وقت که یک بلاکی از حافظه خیلی جدید تا بعدش هم میاره

در برنامه هایی که access patern آنها به صورت sequential است یعنی می‌خوایم به فضایی از حافظه را از این سر تا آن سرش بریم یعنی می‌خوایم یک فضایی از حافظه را دونه دونه پشت سر هم بخونیم در این حالت وقتی یک بلاک miss شد می‌تونیم امیدوار باشیم که اگر برنامه جوری باشه که همین مسیر را بخواد ادامه بده میریم بقیه را هم می‌آوریم چون تناسب access شدن آنها زیاد است. این نوع الگوی دسترسی به خانه های حافظه در instruction ها چیزی کاملا عادی است و به وجود می‌آید ولی در مورد access به داده‌ها درست نیست

به عنوان یک مثال فرض کن access pattern پردازنده به حافظه طوری است که دسترسی به دستورات نظم بیشتری نسبت به داده‌ها وجود دارد

عیب این روش این است که وقتی pattern هیچ نظم (regular) نباشد عملا این prefetcher فقط اضافه هدر می‌دهد چون هیچ نظمی وجود ندارد هر آنچه که می‌آورد و تحویل کش می‌دهد به درد نمی‌خورد

این عیب در این روش فقط در جایی که نظم وجود ندارد نیست. حتی زمانی که در الگوی دسترسی به خانه های حافظه نظم داشته باشیم ولی نظم ها، این نظمی که prefetcher بر مبنای آن کار می‌کند نباشد

Scanned by CamScanner

مثلاً فعلی که prefetcher رضایتی آن کار می‌کند به این صورت است که یک مسیر را از ۵ تا ۵۰ صفحه ولی الگوی دسترسی ما به این صورت باشد که با stride=۲ بخواهد به حافظه دسترسی داشته باشد

Year:..... Month:..... Day:.....

نام‌های دوتا دوتا یعنی ۵، ۷، ۹، ۱۱، ۱۳، ... را بخواند

مثلاً ماتریس سطری با stride=1 خوبه می‌شه ولی ماتریس ستونی با stride=2

یک حالت دیگه اینه که pattern منظم است ولی داره رو به عقب حرکت می‌کند و در این صورت هم روش prefetching به جیغ دردی نمی‌خورد

یک راه کاری که این مشکل آخر را حل کند این است که prefetcher وقتی که یک بلاکی که undemand، miss شده را از حافظه میاره به کش بیاورش کنم هم یک خونه جلوش را بیاورد هم یک خانه بلاکش را بیاورد به صورت کلی اگر n تا بلاکی را هم می‌آورده حالا هم n تا بلاکی را بیاورد هم n تا جلوی

خوبی این روش اینه که از miss ها بلاکی‌ها یکش را به احتمال زیاد می‌خواهد ولی عیب این روش اینه که می‌توانیم مغبون باشیم از اینجا این دو تایی که می‌خواهیم می‌آوریم حداقل یکی را نمی‌خواهیم

اسلاید 8/6

* یک روش prefetching، stride prefetcher است

این روش خیلی‌ای که روش قبل داشت سعی کرده است حل کند و این stride می‌آید مانیتور می‌کند الگوی آدرس‌هایی که recently request شده است در رضایتی آن تصمیم می‌گیرد که چه کاری کند. این آدرس‌ها می‌تواند program counter پردازنده باشد یا آدرس بلوک‌های کش باشد فعلاً که اینجا داریم pc ما همیشه به next instruction address اشاره می‌کند و از طریق pc می‌توانیم بفهمیم الگوی دسترسی به دستورات برنامه به صورت است ولی الگوی دسترسی به داده‌ها را نداریم. اگر بلوک‌هایی که درش داریم ریکوئست می‌شن مانیتور کنیم آدرس‌ها مانیتور، اگر L1 cache باشیم در data cache آدرس‌هایی که به داده داده می‌شه و می‌توانیم مانیتور کنیم آن‌هایی هم که در I cache هستند در I cache می‌توانیم مانیتور کنیم و ترالی آدرس instruction‌ها را بسازیم

یک مقاله که در این حوزه است:

load

در این مقاله می‌گه record می‌کنه آدرس‌هایی که به حافظه می‌آن توسط instruction، یعنی واحدی که به حافظه‌های حافظه توسط instruction load دارن خونده می‌شن. آن آدرسی که دستور load داره generate می‌کنه در خودش ذخیره می‌کنه، recent address که ریکوئست شده را در خودش ذخیره می‌کنه + آخرین آدرسی که دستور load خواسته و از آن n تا آدرس که report کرده (ذخیره کرده) سعی می‌کنه stride درباره یعنی الگو را دربیاره

Scanned by CamScanner

یعنی آیا پردازنده بدون جلو میره، در تا در تا، ها تا ها تا و ... جلو میره، روی عقب می‌ره یا روی جلو

Year:..... Month:..... Day:.....

جلسه ۵

ویدیو اول (Aca_Lec5_chap3_A-1)

اسلاید ۳

- رایج‌ترین روش برای بهبود performance پردازنده‌ها pipline کردن پردازنده است

pipline یعنی اجرای دستورات (یا دستوراتی که هر کدام کام‌های مختلفی دارند) این کام‌ها را با هم overlap کنیم (کام‌های دستورات مختلف با هم هم‌پوشانی داشته باشند)

pipline تحت عنوان موازات در سطح دستورات مطرح می‌شود. در pipline دو دستور به صورت موازی مثلاً با هم اجرا می‌شن بلکه دستورات همچنان به صورت sequential دارن اجرا می‌شن (شیب سرهم) فقط این شیب سرهم زمان فاصله‌اش کمتر شده

- برای مدیریت pipline دو تا approach وجود دارد (برونکرد طراحی):

• برونکرد نخست افزاری که بدون این که نرم‌افزار خودش را درگیر ساختار پردازنده و pipline بودن و نبودن بکند همه‌ی اتفاقات در سطح نخست افزار مانیتور و مدیریت می‌شود. با توجه به این که مدیریت نخست افزاری pipline مستلزم مدارهای کنترلی اضافه هست و سربار انرژی مصرفی به بار می‌آید و پردازنده را پیچیده می‌کند، معمولاً در پردازنده‌های desktop سرور و جاهایی که ما performance بالایی می‌خواهیم از این approache استفاده می‌شود، ولی در پردازنده‌هایی که مثلاً در دستگاه‌هایی که power، area، وزن، هزینه و ... برایشون مهم هست ولی high performance نیستند، در اینجا اندک کاربردی ندارد

• در کنار approache نخست افزاری approache نرم‌افزاری است که کامپایلر مدیریت pipline را به دست می‌گیرد یعنی کامپایلر با علم به این که pipline چطوری عمل می‌کند حوزه‌ی برنامه‌ها را تبدیل به کدهای ماشین می‌کند، دستورات را تا آخر مهم می‌چینند که pipline دچار مهم در دسری نشود و بدون این که به واحد نخست افزاری نیاز داشته باشد بتواند هر دستوری که سرعت را راحت اجرا کند.

این approache نرم‌افزاری جوابی کاربرد ندارد، به خصوص در سیستم‌های عام منظوره به درد نخواهد خورد

اسلاید ۵

* در بحث CPI هدف این است که ما CPI را تا جایی که می‌توانیم کاهش دهیم، کاهش CPI یعنی بهبود performance. چیک پردازنده‌ای که با فرکانس مشخصی کار می‌کند، هر چقدر که معماری و ...

Scanned by CamScanner

وساختار درونی را جوری تغییر دهیم که بتوانیم throughput بیشتری داشته باشد یعنی CPI بهتر شده است.

Year:..... Month:..... Day:.....

می‌دانیم که pipline، throughput را بهتر می‌کند

pipline برای خودش CPI ایده‌آل دارد که تابع چند پارامتر است:

CPI، pipline برابر است با:

- CPI، pipline ایده‌آل (یعنی هیچ مشکل و چالشی برای اجرای دستورات وجود ندارد و دستورات با فاصله گام‌های یک واحد یا شیفت سر هم فاصل دارند و زمان و هیچ نقطه‌ی خالی در pipline وجود ندارد و هیچ دستوری تحت هیچ شرایطی در pipline جلویش گرفته نمی‌شه و pipline کاملاً پر است و utilization آن صد در صد است)

- این CPI را باید جمع کنیم با تعداد stallها یا بازه‌هایی از زمان‌هایی که pipline بخاطر مسائل structural و ساختاری stall می‌شود

- به اضافه‌ی stallهایی که بخاطر data hazard ها پیش می‌آید به وجود می‌آید

- به اضافه stall و متوقف‌هایی که در دستورات به خاطر مسائل کنترلی یعنی انتقالات و branchها که دستورات کنترلی به آنها داخل می‌زند

وقتی می‌گیم دستورات در pipline باید اجرا شوند، داشتن‌ای از دستورات پشت سر هم که در حافظه هستند فاصله‌ها آنجا به اندازه‌ی یک خونه حافظه از هم هست را می‌گوییم pipline نیستیم اما می‌دانیم برنامه‌ی ما این است که همیشه یک مسیر مستقیم را نمی‌رود و بعد از چند دستور که مستقیم رفت می‌پره به یک جای دیگر و از اینجا دوباره چند خط مستقیم می‌ره و دوباره می‌پره به جای دیگه و برنامه داشتم در حال jump کردن از این ور به اون ور هست و می‌دونیم که این پریدن‌ها باعث می‌شود که pipline نتواند این دستورات را پشت سر هم بفرستد به همین دلیل موازات در سطح دستورات در pipline محدود می‌شه به basic blockهای ما

basic block قطعه‌ای از برنامه است که در آن قطعه برنامه هیچ انشعابی ندارد و مستقیم و پشت سر هم دستورات را باید اجرا کند، یک basic block وقتی تمام می‌شود که به یک دستور branch و دوراهی برسیم در اینجا pipline متوقف می‌شود و یا کنید دستور انشعاب مشخص بشه و بعد دوباره pipline را از اول پر کنیم

سایز basic blockها در برنامه‌ها به صورت typical حدود ۳ تا ۶ دستور است که خیلی کم است

Scanned by CamScanner

و pipline که خودش ۴ و ۵ طبقه است تا می‌آید پر بشه و utilization به صد درصد برسه، می‌رسه به یک

Year:..... Month:..... Day:.....

دستور branch و باید صبر کند یعنی ۴ تا دستور می‌رسته، دستور اول داره

خارج می‌شه دستور آخر داره وارد می‌شه که باید جلوی دستور بعدی را بگیرد و صبر کند ببیند دستور branch

می‌خواد کجا بره و این باعث می‌شه که utilization ، pipline خالی پایین بیاد و جلسه‌های

بسیاری از pipline وجود داشته باشد

برای این که بتوانیم از pipline خوب استفاده کنیم باید از تکنیک‌هایی استفاده کنیم که حتی در حالت

خروج از basic block ها هم بتوانیم pipline را مشغول نگه داریم یعنی حتی در مواجهه با

دستورات branch ، pipline را متوقف نکنیم

اسلاید ۶:

یکی از بهترین جاهایی که همیشه موازات در سطح دستورات را implement کرد و ازش استفاده کرد

در loop ها هست

loop level parallelism می‌دونیم هر loop یک branch دارد و این branch ها مزاحمت

ایجاد می‌کنند، در loop ها چون ما کارهای تکراری داریم انجام می‌دهیم و این کارها از قبل مشخص هستند

می‌توانیم با تکنیک loop unrolling کاری کنیم که در هر Iteration کارهای بیشتری را انجام دهیم

و Iteration های مختلف یک loop را در pipline با هم اجرا کنیم

یا استفاده از ساختار SIMD مثل vector processor ها و Gpu ها این امکان را

به ما می‌دهند که Iteration های مختلف loop را با هم اجرا کنیم مثلاً در کدی که در همین

اسلاید وجود دارد ما یک loop داریم که در هر Iteration یک ایندکس از یک آرایه را

می‌خواند و بعد آن عدد را آپدیت می‌کند مثلاً ۱۰۰۰ بار قرار اجرا شده، ما در هر Iteration هیچ کاری

نمی‌توانیم بکنیم برای بهبود performance چون این عملیات باید به صورت sequential اجرا شود

اما چون همه Iteration های این loop مستقل از هم هستند می‌توانند به صورت موازی اجرا شوند

اما چالش pipline کجاست؟ همه چیزی باعث متوقف شدن آن می‌شود

ما مسئله‌ای داریم به نام Data Dependency یا وابستگی داده‌ها

می‌گیم دستور j وابسته است اگر دستور i دیتایی generate کند که مورد استفاده‌ی

دستور j باشد یا این که j به k وابسته باشد و k هم به i در این صورت هم j به i وابسته است

Scanned by CamScanner

مشکلی که این dependency ایجاد می‌کند این است که دستورات وابسته نمی‌توانند همزمان انجام شوند پس نمی‌توانند همزمان وارد pipline شوند و پشت سر هم جلو بروند

Year:..... Month:..... Day:.....

اسلاید ۲۷

Dependency دستورات خاصیت برنامه است پس نمی‌توانیم وابستگی‌ها را از بین ببریم تا pipline خوب عمل کند بلکه باید آنها را مدیریت کنیم، pipline باید این وابستگی‌ها را detected کند و اگر دستوری در حال اجرا دارد و دستوری وابسته به آن می‌خواهد ورود کند باید جلوی ورود دستور جدید را بگیرد و pipline stall شود (همه‌ی dependency‌ها منجر به stall شدن پایپلاین نمی‌شوند و اونایی که باعث‌شون کم است منجر به stall نمی‌شوند)

* چیزهایی که data dependency به ما تحمیل می‌کند:

- می‌تواند منجر به hazard یا خطرات شود

- ترتیبی که نتایج باید براساس آن محاسبه شود چون dependency یک ترتیبی را برای انجام عملیات دارد که حتماً باید فلان چیز قبل از فلان چیز انجام شده باشد

- در ما توان محاسباتی می‌گفت مثلاً ۳۰ درصد قابل موازی شدن نیست و ۷۰ درصد‌ها قابل موازی شدن است و اون ۳۰ ز به دستورات قبل از خودشون وابسته هستند و باید به صورت sequential انجام شن که این ۳۰ دستور در لایه‌ی دستورات برنامه هستند و باید sequential انجام شود این چون upper bound هست به آدم می‌گفت. آدم می‌تونه هر چقدر resource اضافه کنی، هر چقدر پایپلاین را قوی کنی یا هر چقدر multi core کنی تمام یک fraction از برنامه هست که حتماً باید به صورت sequential انجام شود

این dependency‌ها وقتی که operand‌ها داخل رجیستراها هستند کشف کردنشون راحت است ولی بعضی وقت‌ها فهمیدن این که دستورات به هم وابسته هستند آسان نیست مثلاً وقتی که operand‌ها داخل حافظه هستند و آدرس‌های حافظه این dependency را ایجاد می‌کرده‌اند و به صورت آنلاین قبل از اجرای برنامه مشخص نیست که چیزی است و وابستگی دارند

در نتیجه data dependency می‌تواند خوب یا بد و مانع performance خوب از پایپلاین باشد

اسلاید ۸۸

دیاگرام dependency دارم به نام: Name dependency

دو تا instruction که از نام‌های یکسان برای operand‌هاشون استفاده می‌کنند ولی لزوماً وابستگی

Scanned by CamScanner

به هم ندارند می‌تواند برای pipline مشکل ساز باشد

Year:..... Month:..... Day:.....

یک مشکل دیگر dependence Antidependence است که می‌گوید

instruction j می‌خواهد در یک رجیستر یا خانه‌ای از حافظه بنویسد که آن رجیستر یا خانه از حافظه را دستور i می‌خواهد بخواند، دستور i قبل از j است که باید حواسمون باشه که هیچ وقت j قبل از این که i رجیستر را بخواند در آن ننویسد

این dependence برای ما مهم است زیرا ما می‌خواهیم حداکثر استفاده را از پایپلاین بکنیم و به وقت‌های برای حداکثر کردن pipline باید دستورات را جابه‌جا کنیم تا دستورات وابسته از هم دور شوند ولابه‌لای آنها دستوراتی که وابسته نیستند قرار بگیرند و با این جابه‌جایی باید حواسمون باشه Antidependence پیش نیاد و وقتی Antidependence داریم باید ترتیب اجرای j و i را چون جوری نیست انجام بدهیم

یک مشکل دیگر در dependence ها output dependence است یعنی instruction i و j دیتای مشترک ندارند که بخواهند بخوانند یا ورودی یکی خروجی دیگری نیست اما نقطه هر دو یک جاست یعنی هم i می‌خواد رجیستر R را آپدیت کند هم j که بعد از i قرار گرفته در این شرایط باید حواسمون باشه اول i رجیستر R را آپدیت کند بعد چون بعد از این که کار j تموم شد R باید چیزی باشه که j گفته نه چیزی که i گفته بعد این j هم دستوراتی بوده که بخواند از خروجی i استفاده کند

این دو نوع وابستگی، هر دو وابستگی واقعی نیستند و وابستگی هستند که خود instruction آن را ایجاد می‌کند و تقصیر کسی است که برنامه را به دستورات پایپ تبدیل کرده است

راه‌حل چیه؟ از تکنیک‌های register renaming استفاده کنیم یعنی اسم رجیسترها را تغییر دهیم که این کار را یا باید کسی که داره کد سطح بالا را به کد ماشین تبدیل می‌کند انجام بده یا این که کاملاً تخت افزاری مدیریت شود یعنی می‌توان کنترلر پردازنده را دستکاری کرد و پردازنده مطابق محل هر چیزی که برنامه می‌گه نباشه و خودش تغییراتی انجام بدهد

استاد: ؟

dependence داده چه وابستگی واقعی که اول گفتیم چه دو تا وابستگی بالا به تعیم هر سه خاصیت برنامه هستند و وجود دارند که این سه نوع می‌تواند منجر به سه نوع hazard شوند

این hazard ها: چون داده‌ای واقعی است که اگر عوض بخورد باید پایپلاین stall شود

Read after write (RAW) یعنی از جایی که می‌خوام بخونم باید رجیستری بخونم که بعد از یک instruction باشد

Scanned by CamScanner

۲) write after write (WAW) : output dependence عامل آن است یعنی ما دیگر رجیستر می‌نویسیم ما که قبل از آن کسی دیگری باید نوشته باشد

Year:..... Month:..... Day:.....

۳) write after Read (WAR) : Antidependence عامل آن است وما باید جایی بنویسیم اما بعد از این که آن چیزی که محتوای رجیستر است از آن خوانده شده باشد

- غیر از دیتاهایی که بهم وابسته هستند یک مشکل دیگه داریم به نام control dependence که خارج دستورات شرطی و انشعابات به وجود می‌آید در اینجا چیزی که مطرح می‌شه این که دستورات load store و ... را می‌توانیم با دستورات branch هم جابجا کنیم یعنی، آیا می‌توان یک دستور که بعد از یک branch هست را آورد قبل از یک branch و یا برعکس

اگر دستوری را که بعد از یک branch بوده بیاریم قبل از branch قرار دهیم حتی اگر operand یکسان و حساب نداشته باشند و هیچ وابستگی بهم نداشته باشند مثلاً یک دستور داخل if را بیاریم بالای if بیاریم آن وقت دستوری که به صورت شرطی اجرا می‌شده و گاهی اجرا می‌شده گاهی اجرا نمی‌شده الان به صورت غیر شرطی در حال اجرا است و ما هیچ کنترلی روی آن دستور نداریم یا مثلاً دستوری که قبل از if بوده را داخل if بیاریم می‌شه دستور شرطی و اشتباه می‌شه پس امکان جابجایی دستورات با branchها را نداریم

ایده‌ها

یک مثال:

- قطعه کد این صفحه را در نظر بگیریم، خط اول می‌گه $x_2 + x_3$ کن بریز در x_1، خط دوم می‌گه x_4 را با x_5 مقایسه کن اگر equal بودند برو به L، بعد $x_6 - x_1$ کن بریز در x_1 بعد هم branch L می‌گه x_8 با x_1 or کن بریز در x_9، در اینجا add و sub هم وابسته هستند چون sub به x_1 احتیاج داره که x_1 در add هست، دستور beq وابستگی به بقیه نداره اما می‌شه بردش بالای add یا پایین sub چون اگر ببریم بالای add دستور add که باید انجام بشه در صورتی انجام می‌شه که beq به L نپرش نکنه یا اگر ببریم پایین sub، sub یک x_1 تولید می‌کنه که شرطی باید عمل کند اگر beq بیاد پایین sub دیگه شرطی عمل نمی‌کند، دستور or هم به add و sub وابسته است و قابل جابجایی نیست

Scanned by CamScanner

مثال ۳۲

در این قطعه کد دستور sub داره چیزی تولید می‌کنه که بلافاصله در add یا beq مصرف می‌شه و x_4 هم

از اول به چیزی وابسته نیست، اگر دستور sub را ببریم در بالای beq اتفاقی نمی‌افته چون تنها کسی که x_4 براش مهمه add هست و با این جابجایی مشکل برایش به وجود نمی‌خورد (در صورتی اگر ببریم قبل از beq قاعده به صورت شرطی نیست ولی در اینجا مشکلی ایجاد نمی‌کند، با این فرض که قبل از این لاز مقداری در x_4 نبوده)

اما دستور add را نمی‌توانیم جابجا کنیم چون دستور add، x_5 را تولید می‌کند که باید به صورت شرطی باشد

تغییر شرطی، به add اول هیچ کسی وابسته نیست و از x_5 استفاده نمی‌کند، جای sub را برای این جابجا کردیم که وابستگی واقعی بین sub و add بعدش بود

ویدیو دوم (Aca_lec5_chap3-B)

اسلاید ۳۴

* کامپایلر می‌تواند برای این که پایپلاین را پر کند حواس خودش را جمع کند dependence های instruction و باید سعی کند تا جایی که می‌تواند این دستورات به هم وابسته را از هم دور کند

مثلاً: یک آرایه ۱۰۰۰ تایی داریم که قراره همه‌ی خونه‌های این آرایه را با یک s جمع کنیم و آپدیت کنیم

چیزی که کامپایلر راحت پایپلاین می‌کند این که (مثل جدول پایین صفحه) اگر دستور Floating point (FP) در ALU، اگر چیزی را تولید کند و دستور FP دیگری بخواهد از آن operand استفاده کند بین دستور دوم و اول باید سه کلاک فاصله بیفتد

اگر operation، Floating point در ALU چیزی را تولید کرد و کاش چیزی قرار است store شود و می‌خواهد در حافظه بنویسه بین این دستور store و اون operation باید دو کلاک باشد

اگر دستور FP داریم که اتفاقی دارد که باید توسط یک دستور load از حافظه خوانده شود فاصله بین دستور load و FP باید یک دستور باشد

اگر دستور store داریم که چیزی می‌خواد store کند که توسط یک دستور load باید آمده باشد فاصله بین این load و store می‌تونه صفر کلاک باشد

Scanned by CamScanner

Year: Month: Day:

کامپایلر وقتی این دستور در خطی را می‌بیند وقتی می‌خواد تبدیل کند به دستورات پایه اون برنامه اون loop را تولید می‌کند، در خط اول loop ما دستور load را به صورت Floating point انجام می‌ده، بعد در اینجا حتما یی را از آدرسی که در رجیستر x_1 هست ~~می‌خوانند~~ از حافظه می‌خونه در رجیستر F_0 می‌ذاره (رجیستر F_0 رجیسترهای Floating point هستند) در واقع اینجا x_1 را در یک رجیستر می‌ذاره

بعد F_0 که بالا خوندیم با F_2 جمع می‌کنیم می‌ریزیم در F_4، از این دستور می‌تونیم بفهمیم که این s ما در F_2 از قبل بوده (اون d که دستور جمع double باشد)

بعد باید حاصل جمع را در خونه حافظه، از همون جا که دیتا را خوندیم به همون جا برگردونیم پس یک دستور store داریم که می‌گه F_4 را بنویس در خونه حافظه که آدرسش x_1+0 هست

حالا از آدرس x_1، ۸ تا کم می‌کنیم می‌ریزیم در x_1 (عملیات $i=i-1$ را انجام می‌دهد)

حالا x_1 با x_2 compare می‌کنیم اگر مساوی نبود برمی‌گردیم به loop ما (از اینجا می‌تونیم بفهمیم x_2 حاوی صفر است و x_1 حاوی اولین آدرس آرایه‌ای پس x_1 که خونده می‌شه)

در تولید این کد باید حواسش به وابستگی‌ها و خاصیتشون باشه. چون از قبل گفتیم بین دستور Fp و load یک کلاک باید فاصله باشه پس بین Fld و Fadd به اندازه یک کلاک پایپلاین را قفل می‌کنیم و کامپایلر پس این دستورات stall می‌ذاره

در این جا یکدوم دستورات را می‌تونیم جابه‌جا کنیم که stall ها از بین بره یا کمتر شه؟ چیزی که می‌دونیم اینه که دستور load و add و store حتما باید شبیه هم اجرا بشن، دستور bne هم که باید حتما آخر اجرا بشه چون اگر بره بالا اگر دستوراتی که شرطی نیستند، شرطی می‌شن و ممکنه اجرا بشن

پس تنها دستوری که می‌تونه جابه‌جا بشه addi است که می‌تونه بره پایین bne یا بره بالا که می‌شه

قطعه کد سمت راست، ~~[illegible]~~ ما در addi داریم x_1 را آپدیت می‌کنیم که در bne بخش احتیاج داریم

و اگر addi بره اون بالا مشکلی برای bne ایجاد نمی‌شه ولی برای (x_1) در Fsd مشکل ایجاد می‌کنه و وابستگی بین Fsd و addi، WAR است یعنی Antidependence هم دارند

پس می‌تونه addi را ببره بالا ولی با Antidependence حل کنیم پس چون الان با این جابه‌جایی وقتی در دستور Fld، x_1 خونده می‌شه در دستور addi از بین ۸ تا کم می‌شه حالا اگر در Fsd به x_1 ۸ تا اضافه کنیم مشکل حل می‌شه ~~[illegible]~~ الان با انجام این تغییرات ۱۲٫۱۰ درصد Performance بهبود می‌یابد

Scanned by CamScanner

اسلاید ۶

Year:..... Month:..... Day: یک راهکار دیگه که کامپایلر می‌تونه در پیش بگیره loop unrolling است

loop unrolling کمک می‌کنه بتونه کامپایلر انداخت و برنامه‌نویس هم می‌تونه این کار را بکند

یعنی loop unroling یعنی Itration هاش را کم کنیم و در هر Itration کار بیشتری کنیم

مثلا همون مثال قبلی را بیاریم unroll کنیم یعنی به جای این که در هر Itration فقط یک x حساب بشه

چهار تا x حساب بشه و به جای این که i را یه دونه یه دونه کم کنیم ۴ تا ۴ تا کم کنیم

فایده‌اش که unroll کردن loop داره اینه که به ازای هر Itration برای مثال اسلاید قبل دو تا دستور

اگر اضافی بود در ای ۱۲ بودند حالا آن دو تا را سرشکن می‌کنه به ازای هر Itration که انجام می‌ده

پس می‌آیم می‌گیم یک آدرس از x را بیرون بریزد در F0 ، بعد add کن بعد store کن و بعدش branch

نکن برگرد بلکه یک آدرس از x کم کن دوباره load کن دوباره add کن و store ، دوباره

یکی کم کن و دوباره add و store کن ، به دو دیگه هم این کار را بکن و همین اش همه‌ی ۸ می‌کردیم

الان همه‌ی ۳۲ می‌کنیم و i = i - 1 می‌شه i = i - 4 و دوباره برمی‌گردیم به اول loop

نمی‌شه همه‌ی loop ها را باز کرد که اصلا loop نداشته باشیم چون کار برنامه‌نویس سخت می‌شه و تعداد دستورها

با unrolling بیشتر می‌شود

آیا با این کار حافظه کندتر یا سریع‌تر از قبل اجرا می‌شود؟ اون stall هایی که در کد قبلی داشتیم باید اینجا هم بزاریم پس کامپایلر

باید با جابه‌جا کردن دستوراتی که الان دستش باز شده هم شده کلا stall را حذف کند

در این کد دستورات سه تا سه تا به هم وابسته هستند مثلا یعنی سه تای اول هم وابسته‌اند، سه تای دوم هم وابسته‌اند و

با بقیه کاری ندارند و ... پس باید این وابستگی را stall بین این دسته‌های سه تایی از کدها را از بین ببریم پس

می‌آیم بین Fld و Fadd در خط اول و دوم بیاریم Fld در خط چهارم را بزاریم ، بین Fadd در خط سوم و

Fsd در خط چهارم دستور Fadd که در خط پنجم هست را بزاریم و بین Fld و Fadd در خط چهار و پنج دستور

Fld در خط ۷ را بزاریم و ...

اسلاید ۷

در نهایت کدی که کامپایلر می‌ده کد اسلاید ۷ است ، در نهایت ۱۴ کلاک طول می‌کشد و در الان ما تریس با ۳/۵ کلاک

کارش را انجام داده یعنی در هر ۳/۵ کلاک یک x را تو نسبیم اپدیت کنیم

Scanned by CamScanner

اسلاید ۸۸

Year:..... Month:..... Day:.....

branchها مشکلات زیادی برای ما ایجاد می‌کنند و مشکل اصلی پایپلاین این branchها هستند

این جدول می‌گه اگر یک دستور branch رسیدی اون branch اول Fetch می‌شه بعد decode

می‌شه بعد Exe و mem و WB انجام می‌شه، فرض کنیم اینجا ها تا stage پایپلاین باشه

خب ما یک instruction را Fetch کردیم ~~[illegible]~~ بعد که decode کردیم فرمان با این دستور بعدی

Fetche شده، دستور اول که decode شد فهمیدیم instruction، branch است و قرار بود دستور

branch می‌رسیم می‌فهمیم مسیر درواقعی هستیم یعنی دستور ما بیره یا taken بشه یا نیره و not taken

شود و چون نمی‌دونیم می‌بره یا نمی‌بره باید به دستور دوم که Fetch شده نگاه کنیم و ایستا جلو نروم بعد branch

را exe کنیم بفهمیم چیه اتفاقی می‌افته اگر taken شد و قرار شد همین مسیر را ادامه بدهیم تا یا decode

انجام بده و تازه ماشه دستور دوم در Fetch هست دستور بعدی نمی‌تواند Fetch شه و به همین صورت پایپلاین

می‌ره جلو

همین یک فلاک stall شدن به ازای هر branch بین ۱۰ تا ۳۰ درصد سرعت اجرای برنامه را می‌تواند

کاهش دهد و این مقدار زیادی است پس باید کاری کنیم که پایپلاین کمتر خالی branch دست نگه دارد

اسلاید ۸۹

اگر بخواهیم پایپلاین را مجبور کنیم در شرایط branch همه کارش را انجام دهد کن وقت مجبوریم که از روش های

prediction استفاده کنیم. ساده‌ترین راه این است که جلو هایک branch داریم که دو راه پیش رو داره یا

می‌ره یا نمی‌ره به pipline بگیم تمام راهت را ادامه بده و به روی خودت نیار که به branch رسیدی

و اگر branch گفت من باید بروم پایپلاین هزینه‌اش را می‌پردازد و اون دستوراتی که قبل از branch

وارده دور می‌ریزه و از جایی که branch می‌شه (target address) دستور می‌رم دستور می‌خوانم و

میارم. ~~[illegible]~~ در چند درصد مواقع اگر این سیاست را در پیش بگیریم خوشبختانه درست

است چون ما در if و loop، branch داریم و loop ها بخش عمده‌ی زمان اجرای برنامه را

به خودشون اختصاص داده‌اند پس branch هایی که در loop ها هستند مهم‌ترین branch ها هستند

اگر فرض کنیم taken، not taken، else if، ۵۰، ۵۰ باشه در loop قطعاً اینجوری نیست و

branch ها در loop، n بار یک کار را انجام میدن و بار n+1 ام یک کار دیگه را انجام میدن، اون n بار

اول branch ها taken هستند و پس این سیاست که بیاییم فرض را بر not taken بودن بذاریم اصلاً به درد

Scanned by CamScanner

loop ها می‌خوره و شاید بهتر باشه که فرض کنیم همیشه taken می‌شه و وقتی not taken می‌شه هزینه‌اش را بدیم

Year:..... Month:..... Day:.....

* یک راه بین این دو راه وجود داره این که branch های back warde را فرض کنیم taken هستند و branch های Forwarde را فرض کنیم not taken هستند یعنی branch هایی که رو به جلو می‌خوان بیرون بشن IF و else ها هستند و اونایی که رو به عقب هستند branch، loop ها هستند

* روش دیگر این است که از کامپایلر خواهش کنیم از روش Delayed brand استفاده کند که این روش می‌گه ما می‌دانیم هر وقت به branch خوردیم یک تا دو کلاک طول می‌کشه بفهمیم کدوم طرفی می‌خوایم بریم این یک، دو کلاک سلطانی را با دستوراتی که نقش هستیم باید اجرا شوند پر کنیم یعنی از دستورات بالای branch که در branch نیستند برداریم بزاریم بعد branch اجرا شوند و پردازنده هم بیکار نداشته باشه دستوری که به سیدی بخش branch هست یا تاثیر دستوری بعد از branch هست خودم نداشتم رو می‌خوام اجرا بشه و همانجوری به branch سیدی این را متوقف نکن. پایپلاین با علم به این که ما اینجوری برنامه‌ریزی کردیم همیشه یکی دو دستور بعد از branch را هم جزء دستورات داخل branch افتادن می‌شیم و اونا را هم تا آخر اجرا می‌کنه بدون این که قید شرط روی آنها بگذارد و اینجوری سلطانی را پر می‌کنیم و delay branch کاری است که کامپایلر باید زحمت آن را بکشد و با علم به این که پایپلاین به این صورت عمل می‌کند که یک دستور بعد از branch را همیشه بدون شرط اجرا می‌کند باید دستوری پیدا کند که اونجا بگذارد و اگر هیچ دستوری پیدا نکرد از دستور nop استفاده کند

اسلاید 31

پردازنده‌ای که با یک Delay branch کار می‌کند و branch را بدون شرط اجرا می‌کند و برای هر branch یک کلاک بیشتر می‌خواد که تصمیم گیری می‌کند یا نمی‌کند

حالا طبق این اصل به یک دستور branch می‌رسیم که قرار است taken شود، یک دستور هم بعد از دستور branch داریم که بالای branch بوده ولی به نظر این یک کلاک سلطانی آوردیم پایین branch و بعد دستوراتی هستند که داخل branch بوده اند

مراحل اجرا هم مشخص و ساده است. فقط دستورات بعد از exe می‌فهمه که taken نمی‌شه و بقیه دستورات با خیال راحت اجرا می‌شن

Scanned by CamScanner

حالت دوم) دستوری است که taken می‌شه. وقتی دستور اول به exe می‌رسه و مشخص می‌شه که باید taken

Year:..... Month:..... Day:..... Fetch

شه دستور سوم که جایی است که branch به اون می‌ره همزمان با هاش می‌شه.....

پس) دستور branch delay اگر به درستی انجام شود می‌تواند خطا و حباب‌ها را از بین ببرد

ویدئو سوم (Aca-Lec5-chap3-c-1)

سیستم‌هایی که نیاز دارند کامپایلر بحث ویژگی‌های خاص بحث ابزار، نرم‌افزار را دستکاری کند در سیستم‌های خاص منظوره استفاده می‌شود مثلاً یک سوپرکامپیوتر با قدرت زیاد در جای خاصی داره کار مهمی را انجام میده مثلاً پیش‌بینی آب و هوا که داره کار خاصی انجام میده و قدرت پردازش و محاسبات بالایی احتیاج دارد؛ اینجا می‌خوایم که ما هر چیز بدیم برای optimized کردن نرم‌افزار

اما در سیستم‌های عام منظوره با پردازنده‌های عام منظوره این کار به صرفه نیست

اسلاید 84

branch prediction

~~در سیستم‌های عام منظوره~~ یک مسئله‌ی که در روش بالا داره اینه که ممکنه کامپایلر نسبت‌ها نتوانه برای همه‌ی branch ها آن دستورات را پیدا کند و خیلی وقت‌ها آن slot هایی که پردازنده سر در دستم است که چیکار کند اتفاقا باید بیکار بماند

branch prediction یعنی این که به پردازنده بگیم هر وقت به branch رسیدی قبل از این که بدونی چکار کنی یک حدسی را یا تیر یا تیر در پیش بگیر. در واقع branch prediction یعنی پردازنده سر دستور branch توقف نکند و طبق چیزی که پیش بینی می‌کند عمل کند. اگر پیش بینی درست نباشد و ما ضرر کنیم این ضرر و خسارت performance نیست چون بازه‌ای که قرار بوده بیکار بمونه یه کاری انجام داده بیشتر ضرر از منظر انرژی که خاص مشغول کردن CPU و حافظه و ... است

branch prediction ها دو دسته‌اند: 1) Dynamic 2) static

تفاوت این دو این است که در روش‌های static ما یک تصمیم را از اول می‌گیریم و تا آخر برنامه‌ی چون عمل می‌کنیم و درست و غلط هر چیه که بر دعمل کند و حتی اگر در آینده هم متوجه شویم تصمیم ما غلط بوده باید برنامه‌ی چون تصمیم پیش بریم. این روش محبوبیت زیادی ندارد

Scanned by CamScanner

روش های Dynamic به صورت Run time براساس شرایط موجود تصمیم می گیرد، یعنی اتفاقاتی که در

Year:..... Month:..... Day:.....

سیستم رخ می دهد را مانیتور می کند و بر مبنای آن تصمیم بعدی را تنظیم می کند

روش های Dynamic نسبت به static بهتر عمل می کند اما هزینه دارد، static ها کاربرد

دارم را نداریم اما ساده هستند

branch prediction static یعنی ما قبل از این که برنامه را بدهیم وارد سیستم کنیم و در واقعیت اجرا کنیم و

کارش را شروع کند، در زمان کامپایل خود کامپایلر به هر branch که رسید حدس بزند این branch

بیشتر وقت ها می پره یا نمی پره و بر مبنای آن کامپایلر Flag هایی را به دستورات branch اضافه کند

این Flag وقتی وارد پردازنده می شه به پردازنده می گوید که کامپایلر در مورد این دستور چه گفته.

کامپایلر برای این که بداند دستورات به چه سمتی (سمت پریدن یا نپریدن) متمایل هستند باید یک

profiling انجام بده، profiling یعنی یک دور برنامه را به صورت تستی run کند و ببیند دستورات به کدوم

سمت می رن و رفتار را شناسایی کند (با چند بار اجرا) و بر مبنای آن تصمیم بگیرد

واقعیتی که در مورد دستورات branch وجود دارد این است که خیلی biase هستند به یک سمت یعنی

به صورت ۵۰، ۵۰ عمل نمی کنند و همیشه به یک سمت تمایل بیشتری دارند

نموداری که در این اسلاید می بینیم دارد حدود یک روش branch prediction static را نشان می دهد،

محور x تعداد miss prediction است و محور y برنامه های مختلف هستند که در شرایط مختلف

static اجرا کرده اند و بر مبنای آن یک تصمیمی گرفتند و در شرایط مختلف اجرا کرده اند که در بدترین شرایط

miss prediction یک درصد بوده و در بهترین شرایط ۱۸ درصد بوده. این روش عالی است چون با این کار

در بالای ۹۰ درصد مواقع توانست از stall شدن پایپلاین جلوگیری کند. اما همیشه خوب نیست چون لزوماً

همیشه همه چیز به این خوبی پیش نمیره و ما هیچ وقت به این ۹۰ درصد راضی نیستیم و می خوایم بهبود بدیم

اسلاید ۵:

*

می ریم سراغ branch prediction Dynamic، این روش نیاز داره که history ذخیره کند. یک

branch target buffer داریم، branch target address cachel (btb) این بافر برای

رفتار most recent branch ها را در خودش ذخیره کند و بر مبنای آن برای branch های بعدی

تصمیم گیری کند. در آن btb چیزهایی که ذخیره می شه:

آدرس دستورات branch + آدرسی که می رن بهش پره یا جلو پریده + پیش بینی برای دفعه بعد چیه (اگر

Scanned by CamScanner

پریدن است، آدرس جایی که دفعه قبل پریده در جدولمون داریم اگر پیش بینی نپریدن است که دستور بعدی را از pc، Fetch می‌کنیم

Year:..... Month:..... Day:.....

این Table در زمان instruction fetch، access می‌شه چون (یعنی از آدرسی که pc داره مشخص می‌کنه برو از حافظه instruction next را بخوان و بعد dec کن بشینم چیکه)

وقتی در فاز IF به این Table دسترسی پیدا می‌کنیم چه اطلاعاتی می‌ده و کجا را می‌خوایم access کنیم؟

ستون سمت چپ این جدول branch address را نشان می‌دهد. pc ما دارد next Instruction را مشخص می‌کند که از کجای حافظه خوانده بشود. اگر دستور بعدی ما دستور branch باشد و این دستور branch recently access شده باشد و آن دستور branch که رفتیم بیاریم، خودش به اعماق دسترسی را در این جدول داریم و چون آدرسش را داریم قبل از این که دستور branch از راه برسه ما در این جدول می‌توانیم بفهمیم که دستوری که خواهد آمد دستور branch است. فرض کنیم که با استفاده از pc (pc یک آدرس است) این آدرس را در این Table در ستون سمت چپ سرچ کردیم و پیدا کردیم، حالا در فیلد دوم که Target address است به جای سراغ این دستوری که داره می‌آید دستور branch است و اگر خواد بپره به فلان آدرس حافظه می‌خواد بپره و وقتی خودش آمد دستور بعدی که می‌خواد بیاره با خودش یعنی می‌خواد به اونجایی که بپره آدرسش در Target address است. فیلد prediction bit می‌گه آن دستور که داره میاد branch هست و اگر بپره به فلان جا می‌پره آیا الان گذاشتید بر مبنای پریدن عمل بکنیم یا نپریدن

prediction bit بر چه اساسی مشخص می‌شود؟ کام بعدی که هر کسی می‌خواد بزاره بر مبنای آخرین کاری که برداشتن تقسیر می‌کنیم

اسلاید 36

* پس ساده‌ترین branch predictor در روش‌های Dynamic branch prediction، one bit predictor یا predictor یک بیتی است که این یک بیت به این صورت تنظیم می‌شود: هر وقت که یک دستور branch را می‌بینیم یعنی IF و dec می‌بینیم که branch هست، این دستور branch چون تازه اومده باید در آن جدول insert بشه و وقتی که insert کردیم اون prediction bit را تنظیم می‌کنیم اینجوری که اگر الان پریدن آن بیت را یک می‌کنیم اگر نپرید صفر می‌کنیم و دفعه‌ی بعدی اگر دوباره به این دستور branch رسیدیم به رفتار قبلی آن رجوع می‌کنیم می‌گیم این قبلاً آمده به فلان جا می‌خواست

Scanned by CamScanner

بپرد و نپریده پس به همین راحتی prediction را انجام میدهیم. پس به ازای هر branch که می‌آید اگر جدول پیدایش کردیم یک پیش بینی هم در مورد آن داریم و چون پیش بینی در مورد آن داریم بر مبنای آن عمل می‌کنیم اگر پیش بینی اشتباه بود آن را آپدیت میکنیم

Year:..... Month:..... Day:.....

اینجا state machin آن بیت را هم کشیده، یک بودن را به عنوان Taken در نظر گرفته و صفر را به عنوان not taken

این روش ساده هست و می‌توانیم انتظار داشته باشیم که دقت آن نسبت به static بهتر است

اسلاید ۷۵

روش بالا اگرچه خوب است اما باز ما را راضی نمی‌کند چون سناریوهایی وجود دارد که این روش خوب عمل نمی‌کند

مثلاً یک loop را در نظر بگیریم، آیا این predictor یک بیت در loop ها خوب عمل می‌کند یا نه؟

فرض می‌کنیم برنامه می‌آید و می‌آمد در loop، اولین بار که در loop ما می‌افتد قبیح شناختی از آن نداریم ولی دوم مثلاً چه جوری عمل کرده و چون اصلاً قبلاً وجود نداشته این دستور branch که مربوط به آن loop هست جدول وجود ندارد پس زمانی که براساس pc سرچ می‌کنیم آن آدرس را پیدا نمی‌کنیم و نمیدانیم دستور branch هست یا نیست و دستور branch می‌رسد و pipeline متوقف می‌شود چون دفعه اول است که داره دستور branch اجرا می‌شه. loop ما یک رفتاری از خود نشان می‌دهد که تا n بار بعد هم آن تکرار می‌شه پس اولین بار که دستور branch در loop می‌رسیم در آن جدول insert می‌کنیم و تا زمانی که در loop ما می‌چرخیم براساس آن جدول پیش بینی‌ها درست انجام می‌شه. به جز آخر که دستورات branch در loop تا n بار taken می‌شن و یک بار not taken می‌شوند

آیا این predictor یک بیت اینجا درست عمل می‌کند؟

دفعه‌ی اول اشتباه پیش بینی می‌کند چون هیچ history ندارد، n-1 بار بعد پیش بینی درستی می‌کند، آخرین Ltresion اشتباه پیش بینی می‌کند چون اول پیش بینی می‌کند که می‌خواد برگرده از اول انجام بشه اما خارج می‌شه. پس از n بار، ۲ بار اشتباه می‌ره مسیر را و n-1 بار درست. دقت ما ۹۹.۹ درصد است و ما را راضی می‌کند اما این صرفاً برای loop ها که بدون هست خوب است و برای loop های تودرتو چیزی عمل می‌کند؟

در loop های تودرتو به ازای تمام loop های داخلی این predictor همیشه ۲ بار اشتباه می‌کند و اولین loop داخلی اگرچه قبلاً اجرا شده اما زمانی که بخواهیم دوباره اجرا کنیم چون دفعه‌ی یکی از آن پریده بیرون

Scanned by CamScanner

به آن رجوع می‌کند و اشتباه می‌کند و وقتی در loop داخلی می‌آییم پیش‌بینی برمبنای پریدن است و در سطح بعدی آخر اشتباه می‌کنیم. پس به ازای loop داخلی ۲ بار اشتباه می‌کنیم n-1 بار درست عمل می‌کنیم

Year: Month: Day:

اگر loop خارجی ۱۰۰ بار عمل کند و loop داخلی ۱۰۰۰ بار ما ۲۰۰۰ بار اشتباه عمل می‌کنیم که این برای ما به عنوان معمار کامپیوتر درست نداریم این ۲۰۰۰ بار را بزنیم و درست داریم یک predictor طراحی کنیم که بهتر عمل کند و این ۲۰۰۰ تا را کم کند و اینجاست که ما می‌رویم سراغ predictor دوبیتی که از miss predictor های دوتایی که جلوتر nested loop رخ داده است را از بین ببریم

predictor دوبیتی می‌گوید که این طور نباشد که یک شرط یک رفتار را شرایط داشته باشد است اگر فقط یکبار برای آخرین بار آن رفتار را شکست دهد آنچه که راجبش پیش‌بینی می‌کردیم را بهم بزنیم و برمبنای آن آخری عمل کنیم. یک شرط n بار یک رفتار داشته باشد آخر یک رفتار دیگر داشته باشد یا عادلانه است که برای پیش‌بینی رفتار بعدی به چون بار آخر استناد کنیم و n بار قبلی را نبینیم. پس predictor دوبیتی می‌گوید فقط از آن آخری فاصله بگیر و به چیزی ذره عقب‌تر هم نگاه کن

پس predictor دوبیتی می‌گوید اگر کسی قضاوت من را در مورد یک دستور branch عوض کند باید حداقل دو بار یک رفتار از آن ببینم تا برمبنای آن بخواهم رفتار کنم و شروع براساس رفتار قبلی عمل می‌کنیم یعنی باید دو بار اشتباه کند تا رفتارش را عوض کند

اسلاید ۸۶

state ماشین predictor دوبیتی. دقت کنید که weakly می‌داریم اما دیگر برعکس اگر چون رفتار را داشته باشد می‌گیم رفتارش مشخص شد پس می‌ره در strongly مثلا اگر دو بار پریدیم مبنای این براساس پریدن است پس می‌ره در strongly Taken و اگر دو بار نپرید می‌ره در strongly not taken

وقتی در strongly Taken باشد راحتی به strongly not taken نمی‌رود و برعکس

مثلا تا زمانی که Taken هست در strongly می‌مونه حالا اگر not taken شد weakly Taken می‌ره اگر باز هم not taken شد می‌ره در weakly not taken اگر باز هم نپرید می‌ره در strongly not taken

پس این اسلاید یک مدل دیگه از این state ماشین هست می‌گه اگر یک بار نپرید و اشتباه کرد مثلا weakly taken و اگر باز هم نپرید مستقیم برو به strongly not taken

Scanned by CamScanner

* مشکلی که حالت دوم به اول داره این که در حالت اول ممکنه اون وسط اشتباه کنیم و برای branch هایی که یکی در میون Taken و not taken می‌شن اصلاً چیز خوبی نیست

Year:..... Month:..... Day:.....

خوبی حالت دوم نسبت به حالت اول خیلی در branch هایی که یکی در میون Taken و not taken می‌شن همیشه اشتباه نمی‌کنه ولی همیشه هم درست جواب نمی‌زند و ۵۰ درصد درست عمل می‌کنه، ۵۰ درصد اشتباه

اسلاید ۹

* BTB (Branch target buffer)

تقسیم یک Table داریم که n تا ردیف داره هر بار به ازای یک یک دستوراتی که Program Counter داره از حافظه Fetch می‌کنه ما اون Table را داریم search می‌کنیم تا ببینیم آیا این آدرسی که pc نشون داده است در این Table هست یا نه. و اگر هست برویم ببینیم به کجا می‌خواد بپره و آیا می‌خواد بپره یا نه

در کش مانند Fully associative را دوست نداشتیم. در اینجا در مورد این که چه جوری این search انجام می‌شه و چه جوری pc را می‌تونیم پیدا کنیم چیزی نگفتیم ولی می‌دونیم که سرچ کردن Table به این راحتی نیست. این Table اگر شبیه کش Fully associative باشد که در هر ردیف هر چی بخواهد قرار داشته باشه بنابراین به ازای یک یک instruction هایی که Fetch می‌کنیم باید Table را کامل سرچ کنیم و compare کنیم pc را با آدرس branch ها چون جوری که در کش Tag را با آدرس Tag در ورودی compare می‌کردیم اگر direct متوالی compare داشتیم در compare k ، k-way داشتیم و در Fully چه باید compare می‌کردند). در پردازنده‌ها یک کش نداشتیم، کشی که بیشترین اهمیت پیدا می‌کنه ما چند تا entry در آن جدول نداریم، دو تا، ۴ تا، ۸ تا، ۱۰۲۴ تا ...

و قطعاً نمی‌تونیم بزاریم کنیم چون سرچ کردن آن سخت است و جای زیادی اشغال می‌کند و کوچک هم کنیم history کم می‌شه

برای این که سرچ کردن جدول راحت شود و جدول نیاز به سرچ کردن نداشته باشد و ما به ازای هر آدرس pc یک جای مشخص را به دنبال آدرس branch بفرستیم می‌گوییم branch target buffer را جایگزین کنیم یا در کنار BTB ، branch history table داشته باشیم. bht می‌شه به جای این که آدرس دستور branch را به عنوان یک خانه آن جدول ذخیره کنیم، آدرس دستور branch را استفاده کنیم برای آدرس دهی به آن جدول یعنی اگر جدول اگر مثلاً ۱۰۲۴ تا entry داره، ۱۰ بیت نیاز داره برای آدرس دهی. ۱۰ بیت پایین آدرس branch در خانه‌ی خاص (۱۰ بیت پایین) جایی بشه که یک دستور branch اطلاعاتش در این

Scanned by CamScanner

جدول ذخیره می‌شه پس خود آدرس دستور branch را ما ذخیره نمی‌کنیم بلکه در این جدول ستون سمت چپ
حذف می‌شه، ستون وسط که می‌شه آدرسی که بهش می‌پریم ذخیره می‌کنیم و در

Year:..... Month:..... Day:.....

بیت prediction هم داریم، هر وقت که pc می‌ره یک دستور جدید از حافظه بخونه، ۸ بیت پایین pc را
نگاه می‌کنیم با اون ۸ بیت pc جدول را آدرس دهی می‌کنیم مثلاً ۸ بیت پایین pc اگر شماره‌ی ۱۲۵ را داشت
می‌ریم در ردیف ۱۲۵ جدول را نگاه می‌کنیم در ردیف ۱۲۵ ممکنه چیزی باشه یا خالی باشه، این چیزی که هست آدرس
هدفی است که اگر این دستوری که می‌خوایم بیارم دستور branch هست به اونجا بپره و پیش بینمون هم در فیلد بعدی
مشخص است

در branch history table دو عامل برای این که miss prediction رخ بدهد وجود دارد:

۱) دستور branch در این جایی که ما رفتیم سراغ جدول هست و پیش بینمون این است که می‌پره ولی واقعی
دستور branch می‌آید نمی‌پره و پیش بینمون غلط است

۲) یک پدیده دیگه هم باعث می‌شه که ما پیش بینمون اشتباه باشه و اون پدیده در branch target buffer
وجود نداشت. ~~[illegible]~~ ما داریم می‌گیم در جدولمون که مثلاً ۱۰۲۴ تا entry دارد
با ۸ بیت پایین آدرس داریم آدرس دهی می‌کنیم، نیاز برنامه‌ی ما ۱۰۰۰ است پس ۹ هزار تا دستور
داخل برنامه‌مون هست که ۸ بیت پایینشون مثلاً عدد ۱۲۵ است و به ازای تک تک این دستورات می‌ریم
سراغ این بافر و خط ۱۲۵ را نگاه می‌کنیم اولاً از اون ۹ هزار تا که حداقل به ۱۲۵ خیلی‌هاشون دستور branch
نیستند ولی برای همشون می‌ریم یک پیش بینی می‌کنیم که واقعی ۱ درصد اگر branch بود این کار را بکن و برای
این دستورات اصلاً نیاز نبود سراغ بافر بریم و وقت خودمون را صرف کردیم اما یک مسئله دیگه که وجود داره
از اون چند هزار تا دستور که ۸ بیت پایینشون ۱۲۵ هست چندین یا چند ۱۰ یا چند ۱۰۰ دستور branch
هم لابه لاشون هست در کش برای این که این تداخل را قائل بشیم tag می‌ذاشتیم که بدونیم این دیتا را از کجا
آوردیم اما اینجا شئی وجود نداره، ما یک دستوری را از خونه‌ی ۱۲۵ آوردیم و branch بود در جدول قرار دادیم
و پریده و پیش بینی برای دفعه‌ی بعد هم این است که بپره، چند کلاک بعد یک دستور دیگه‌ای که اون هم در
خونه ۱۲۵ نیست ولی ۱۲۵ قسمت پایین آدرسش هست و آن هم یک دستور branch هست، آن را
داریم میاریم می‌ریم سراغ جدولمون، جدول می‌گه این دستور اگر دستور branch هست دفعه‌ی قبل پریده
ولی ما داریم در مورد یک دستور branch دیگه صحبت می‌کنیم فقط تنها تداخلشون در ۸ بیت پایین آدرسشون
هست که یکی را بیت و این باعث می‌شه که ما اشتباه کنیم، احتمال این پدیده چقدر است باید تست کنیم و ببینیم
در برنامه‌های مختلف چقدر احتمال دارد رخ دهد. احتمالش خیلی زیاده اما چیزی که می‌تونه امیدوارکننده باشه

Scanned by CamScanner

Locality است. درسته که ما می‌گیم ۱۰۰۰ تا دستور branch داریم که ۱۲۵ تا شون آدرس‌شون یکسان است ولی این ۱۰۰۰ تا

Year:..... Month:..... Day:.....

اجراشون یک Locality دارد یعنی یک دستور branch که الان اجرا شد

به احتمال زیاد بعداً هم خودش اجرا می‌شه مثل وقتی که در loop می‌افتیم. این Locality

می‌تونه مشکل درست رو حل کند

یک جدول با ۴۰۹۶ تا Entry، ما الان با فرض این که branch history Table داریم و

مجبور نیستیم Table را سرچ کنیم و هر آدرسی را مستقیم فقط یک جا را دنبال prediction می‌کردیم می‌توانیم

جدولمون را برداشته باشیم، این جدول همان جدول direct mapped است فقط آدرس‌دهی مستقیم می‌خواد

پس entry می‌تونه زیاد باشه. یه جدول ۴۰۹۶ تایی miss prediction rate در

برنامه‌های مختلف تست کردن که از ۱ تا حتی ۱۸ از هم بوده و اگر جدول را هم بزرگ کنیم و تا بی‌نهایت

هم ادامه بدیم چیزی عاید ما نخواهد شد

اسلاید ۱۰:

این نمودار، نمودار هیت که دقت یک prediction دوبیتی با ۴۰۹۶ مدخل را نشان می‌دهد و می‌بینیم که

چقدر دقت این روش بالاست. این نمودار دقیقاً همون نمودار static است در صورتی که باید این از static

بهتر باشه که خوب شون کمیده یه جایی اشتباه شده. خیلی بحث توجه بکنیم فقط می‌خواد بگه چقدر این prediction

خوب است

اسلاید ۱۱:

این نمودار همون نمودار اسلاید قبل هست فقط داره مقایسه می‌کنه predictor ۴۰۹۶ تایی را با predictor

که entry های آن بی‌نهایت است. با توجه به این نمودار درصد miss prediction در حالتی که entry ها بی‌نهایت

باشه با حالتی که ۲ بیتی باشه تقریباً یکسان است. این نمودار نشون می‌ده زیاد کردن سایز Table از یک جایی

هیچ فایده‌ای ندارد و فقط هزینه ایجاد می‌کند و از نظر performance هیچ فایده‌ای ندارد

اسلاید ۱۲:

آیا هر کسی رهتاری که الان داره فقط تابع رهتار گذشته خودش هست؟ یا علاوه بر اینکه تابع رهتار گذشته است

تابع عوامل دور و بر ها هم هست؟

واقعیت این که ما تحت تأثیر عواملی که در محیط رخ دهد تصمیمی که می‌گیریم ممکن در گذشته اصولاً آن تصمیم را

شریک باشیم. branch ها هم همین طور هستند. branch ها در واقعیت به هم برای correlated

هستند یا به هم وابسته اند مثلاً یک branch دفعه‌ی قبل پریده آیا این دفعه می‌پره یا نه؟ می‌گیم چون

Scanned by CamScanner

دیدیم که قبلی پیچیده این دفعه مهم می‌بره اما یک branch وقتی تحلیل می‌کنیم به شرط داره که اون شرط تحت تأثیر عوامل مختلف تغییر می‌کند (شرط تغییر می‌کنه این operand که داخل شرط هست تغییر می‌کنه)

Year:..... Month:..... Day:.....

وقتی که یک branch می‌رسیم اگه مهم است در صورتی پیش بینی کنیم این است که مثلاً چهار بار ده تا
به اضافه این که ما چه جوری رسیدیم به این branch. یک برنامه مسیرهای مختلفی را طی می‌کند تا به یک
نقطه برسه، یک function از جاهای مختلف ممکنه call شود و یک branch ممکنه از مسیرهای branchهای
متفاوتی آمده باشه و این که ما از چه مسیری به یک branch رسیدیم مهم است برای این که بتوانیم رفتار
اون branch را پیش بینی کنیم پس برای branch های corelated علاوه بر رفتار گذشته باید به
مسیری که branch از آن آمده لازم داریم و وقتی مسیرها متفاوت باشند پیش بینیون هم متفاوت است.

پس ما m تا branch recently executed را نگه می‌داریم یعنی کاری به آدرس‌هاشون نداریم، branch اخیر که در جاهای مختلف حافظه آمده‌اند
و taken شدن یا not taken شدن‌شون را نگه می‌داریم. برای این که در مورد یک branch بخواهیم قضاوت کنیم در ابتدا این branch history Table را history
را ایندیکت می‌کنیم یا فیلد مشخصی از آن branch Table را انتخاب می‌کنیم

پس در حالت کلی اگر ما مثلاً یک predictor یک بیتی یا دو بیتی داشتیم اون predictor، (m,n)
بیت داریم که اون n همون یک بیتی، دو بیتی بودن را مشخص می‌کند و اون m می‌گه ما تا چند branch
اخیر را می‌خواهیم تا مسیرها را در تصمیم‌گیریمون لحاظ کنیم. در branch predictor corelated برای هر
branch به جای یک فیلد در Table، 2^m تا فیلد در Table داریم یعنی 2^m تا پیش بینی داریم
برای هر دستور branch. به ازای تک تک این 2^m حالت ما یک prediction داریم. حالا این که کدوم
را انتخاب کنیم اون n مشخص می‌کند

در حالتی که ما m را صفر نداریم همیشه این که ما فقط یک فیلد table داشته باشیم و predictor هم
دو بیتی باشد

اسلاید 13

مثال از branch به حالت corelated است:

فرض کنیم این سه if را داریم. اون if سوم می‌گه اگر aa مخالف bb بود یعنی اگر هم
if اول اجرا شده و if دوم اجرا شده. پس این if سوم رفتارش پیش بینیم ربطی به رفتار گذشته‌ی
خودش نداره بلکه آن چیزی که مشخص می‌کند این if سوم می‌ره یا نمی‌ره رفتار دو تا if
بالاتر است یعنی بستگی به آخرین اجرای دو تا if بالا نگاه کنیم کنیم به جای این که به آخرین اجرای خودش

Scanned by CamScanner

نگاه کنیم. اگر هر دو if اول وارد شده باشند و یا هر دو if وارد نشده باشند if سوم وارد می‌شود. اگر یکی وارد شده یکی وارد نشده اون if سوم وارد می‌شه

Year:..... Month:..... Day:.....

حالا در پایین اسلاید اومده اون if ها را باز کرده به صورت اسمبلی نوشته و مفهومی همون توضیح بالاست که b_3 تابع b_1 و b_2 وابسته است

سمت راست اسلاید یک مثال یکم ساده‌تر برای ما آورده که یک branch فقط به یک حالت قبل از خودش وابسته است.

اینجا if دوم وابسته به if اول است و وابسته به رفتار اخیر خودش نیست

این داره می‌گه ما نباید غافل بشیم از رفتارهایی وابستگی که branch ها به هم دیگه دارند و می‌تونه در پیش بینی ما تاثیر بذاره باشه

اسلاید ۱۴:

* یک مثال از correlated branch:

فرض کنیم که می‌خوایم برای prediction دو بیتی و برای رفتار اخیر دو تا branch که رخ داده تصمیم بگیریم یعنی وقتی branch سوم اول بیاد دو تا branch قبل از خودش چیکار کردند و برای این چهار تا حالت 00، 01، 10 و 11 ایجاد می‌شه و برای یکی از این چهار تا که رخ داده ما سراغ branch table می‌رویم. پس ما در branch correlated یک global history داریم که همون رفتار اخیر branch ها هست به اضافه این که یک local history داریم که برای تک تک branch ها در حالت خاصی از این مسیرها یک پیش بینی داریم

در نهایت یک جدول به صورت جدول سمت راست این اسلاید داریم، جدولی که ۴ قسمت شده و ما هر بار که می‌خوایم پیش بینی کنیم اول پیش بینی می‌کنیم که از بین این چهار تا جدول کدام را انتخاب کنیم که این را global هست که مشخص می‌کنه (بر اساس رفتاری که دو تا branch اخیر داشتند).

هر کدوم از جدول‌ها یک branch history table برای این دستورات خاص هستند که بر اساس پیش بینی که کرده این دستور افتاده در خونه‌ای که با "XX" نشون داده، اینجا عکسه پیش بینی ما درست باشه یعنی غلط باشه، اگر درست باشه یعنی هم global درست عمل کرده هم این predictor دو بیتی برای رفتار اخیر branch درست تصمیم گرفته، اما اگر پیش بینمون غلط باشه دو حالت داره که یا این predictor دو بیتی اشتباه کرده یا این که به این درست انتخاب کرده ولی اون prediction داخل جدول اشتباه بوده پس ما الان باید اون global را آپدیت کنیم از prediction داخل جدول و

۸۱ ما روی global نسبتی نداریم که بخواهیم آن را آپدیت کنیم چون این دو بیت از رفتار اخیری که دو branch قبل از

Scanned by CamScanner

این branch داشته و ما اصطلاحی انتخابی نداریم پس ما باید بیاییم prediction داخل جدول (همون خونه که با XX مشخص شده) را آپدیت کنیم

Year:..... Month:..... Day:.....

پس برای درس دهمی این ۰ است پس درس با این ۲ بیت انتخاب هم می‌چسبند ۱۲ بیت تحویل می‌دهد پس ۴۰۹۶ entry داریم که به ۴ تا ۱۰۲۴ تا تقسیم شده که از بین اینهایی که انتخاب می‌شوند بر مبنای دو بیت پرارزش درس هست

اسلاید ۵ :

مهم: اگر branch predictor داشته باشیم که (m و n) آن فلان مقدار باشد چند بیت دارد؟

مثلاً مثلاً prediction ما بر مبنای predictor دوبیتی است، اگر ۴۰۹۶ مدخل داشته باشیم مثلاً چند بیت خرج کردیم؟

(فرمول کلی = $2^m \times n \times$ number of prediction entry)

predictor دوبیتی ساده که بدون global history باشد و ۴۰۹۶ مدخل داشته باشد

$2^0 \times 2 \times 4K = 8K$ ← پس هر entry دو تا بیت prediction دارد

حالا اگر بین یک predictor (۲و۲) داریم (related است) با همون ۸K که بالا خرج کردیم چند entry خواهیم داشت؟ با توجه به این ۲ ما چهار تا جدول داریم (2^2) و هر جدول هر دو بیت ۲ بیت داره حالا می‌خواهیم بفهمیم چند تا entry داشته باشیم که در مجموع ۸K خرج کنیم

$2^2 \times 2 \times x = 8K \rightarrow x = 1K$

اسلاید ۱۶ :

این نمودار سه حالت را با هم مقایسه می‌کند: (برای branch های دیگه داری نداریم)

حالت ۱) predictor دوبیتی بدون global (نمودار قرمز رنگ)

حالت ۲) با همون حالت قبل وقتی بیایم unlimited باشد (نمودار سبز رنگ)

حالت ۳) وقتی که همون مقدار بیت خرج کنیم ولی به جای این که ۴۰۹۶ مدخل داشته باشیم ۱۰۲۴ تا entry در چهار جدول داشته باشیم (نمودار آبی رنگ)

حالا می‌خواد ببینه miss prediction چند درصد واقع درست از آن نموده

حدوداً همه اول (nasav) حالت ۲ که می‌شه نمودار سبز رنگ به صفر رسیده و آبی قرمز با هم تو بالا می‌هستند

Scanned by CamScanner

در برنامه دوم که اصلاً miss prediction صفر بوده

Year:..... Month:..... Day:.....

~~در سومی هم آبی و قرمز مساوی هستند و سبز صفر است~~

در چهارمی هر سه مساوی هستند، در پنجمی سبز و قرمز مساوی هستند و آبی بهتر از قرمز و سبز است و ...

(هدف اصلی مقایسه آبی و قرمز است)

پس ما می‌بینیم با چون هر سه‌ای که برای حالت predictor در نظر گرفته بودیم اگر correlated در نظر بگیریم صفر نشدیم ولی می‌توانیم در دو اتفاق سود کنیم و این مطلوب است که بدون هزینه‌ی اضافه به دقت بالاتری برسیم و سیستم بهتر عمل کند

اسلاید ۱۷:

* یک branch predictor دیگه داریم بحثش همین روش Tournament

یک Tournament بین حالت global و local.

ما می‌بینیم که branch ها correlated هستند ولی فرض ما همیشه این نبوده و خیلی وقت‌ها بهتر است که صرفاً فقط بر اساس رفتار اخیر خود branch قضاوت کنیم مثل وقتی که در loop می‌افتیم و هر branch ضمناً به رفتار خودش وابسته است

Tournament می‌گه بیا هر دو حالت را داشته باشیم و بین اینها select کنیم که کدام را داشته باشیم یعنی وقتی که یک branch می‌رسیم که دو بیت داده برای branch های دیگه هم در مورد ش قضاوت شود از حالت global استفاده کنیم و وقتی به branch می‌رسیم که دو بیت داده در کنار دستش قرار گرفته و دو بیت داده فقط برای رفتار اخیر خودش قضاوت شود از حالت branch predictor اول استفاده کنیم

پس این Tournament می‌گه اگر دیدید افتادید در predictor که global است و خوب عمل نمی‌کند بروید از اون یکی استفاده کنید و اگر انجام دادید خوب عمل نمی‌کند برگردید همین جا. پس به صورت dynamic خودش را وقتی می‌دهد با شرایط که از چه predictor استفاده کند

موقعی که ما یک ترکیبی از ایلگوریتم‌ها داریم که رفتار قضاوت دارند باید آنها را خودمون را هر بار دم و دمش بدهیم

در روش global رفتارهای اخیر branch ها را مد نظر می‌گیریم (branch ها که آخرین بار رخ داده‌اند)

دو رشته‌ای از بیت‌ها را نگه می‌داریم که یعنی اون branch پریده یک یعنی پریده

در ضمن اون یکی از Table ها را انتخاب می‌کنیم

Scanned by CamScanner

در حالت local کاری به بقیه Table ها نداریم

Year:..... Month:..... Day:..... در نهایت ما یک منطق مثل مثل این اسلاید داریم که ما یک local predictor

داریم یک global predictor و یک global history داریم از آنجایی که تا الان اتفاق افتاده در صورت دستور

branch در جنبای آن ما تصمیم می‌گیریم که global را انتخاب کنیم یا local

اسلاید ۱۸:

حالا برای این طراحی که بین predictor های local و global یکدیگر را انتخاب کنیم یک state machin

داریم که می‌تونه هر یکدیگر دو بار اشتباه کردند برود سراغ اون یکی

اسلاید ۱۹:

این نمودار می‌خواد بگه در این Turnament چند بار local انتخاب شده چند بار global

در برنامه‌های بالای نمودار بیش از ۹۰ درصد مواقع local را انتخاب کرده است و در بقیه‌ی موارد هم تقریبا ۵۰،۵۰ بوده

اسلاید ۲۰:

* در این نمودار می‌خواد بگه در predictor size های مختلف یعنی با سایزهای جدول های مختلف این سه branch predictor چه جوری عمل کردند

چیزی که در مورد miss prediction می‌بینیم سه تا پیدا است. یکی این که وقتی سه سایز جدول را بزرگ می‌کنیم برای هر سه miss prediction کمتر می‌شه و دقت بالا می‌ره

نکته دوم اینه که حالت correlated از حالت local بهتر است اما حالت بهتر از اینها اینکه بین حالتی است که بین اینها باشه و حالت Turnament است

سه چیز اینه که این نمودار داره اشتباه می‌شه (هر سه) یعنی از یه جایی به بعد بزرگ کردن هیچ فایده‌ای ندارد

ویدئو چهارم (Lec5_chap3_D)

اسلاید ۳:

* Dynamic scheduling:

یعنی در حین اجرای دستورات Rearrange کنیم یا ترتیب اجرای آنها را هم بزنیم. بنابراین با تعیین

این که منطق برنامه هم عوض نمی‌کند و روند تولید داده‌ها همون روند خودش را حفظ خواهد کرد باید این امکان را

به سخت افزار بدهیم که بتواند دستوراتی که می‌تواند زودتر اجرا شوند را زودتر اجرا کند هر چند که نوشته شده

باشند

Scanned by CamScanner

مزیت: دیگه کاهیابی را در یک مرحله‌ی اجرای دستورات نمی‌کنیم. مثلاً فهمیدیم کاهیابی مثلاً توی لگم هو تشمندی رخ داده و

Year: Month: Day:

دستورات را موری تنظیم کنه که منجر به حافظه نشود و دستورات وابسته را

از هم فاصله بده که این کاهیابی را خیلی وابسته می‌کند به microarchitecture یک پردازنده در حالیکه یک

پردازنده مبتنی بر microarchitecture از ساختار داخلیش خبر نداشته باشد

حتی اگر کاهیابی هم از microarchitecture خبر داشته باشد خیلی از وابستگی‌ها می‌تواند وجود داشته

باشند که در زمان کاهیابی معلوم نیست که وجود دارند یا نه و در Run time مشخص می‌شه. در این موارد

سخت افزار می‌تونه کاری کنه ولی نرم افزار از قبل هیچ کاری ازش برنمیاد

عیب: هم این دو مورد از سخت افزار و مدارهای کنترلی باید بزنیم و exeption هایی که در حین اجرای

دستورات ممکن رخ بده و حتی دستورات خارج جا مشین مدیریت آنها سخت می‌شود

اسلاید ۵:

* Dynamic scheduling دو نوع داره:

۱) اجرای خارج از ترتیب (out of order execution)

۲) completion خارج از ترتیب (out of order completion)

دستورات Fetch، decode، exe می‌شود بعد Result آن را می‌نویسیم. این روش execution

همون فاز execute شدن بعد از decode شدن یک دستور است

completion شدن زمانی است که Result نتیجه گرفته است در مقصدهایی که write

در Dynamic sche می‌تونیم دستورات را خارج از ترتیب اجرا کنیم ولی به ترتیب ترجیح کنیم به حالت

دیگه هم اینکه اجازه بدیم خارج از ترتیب ترجیح هم بشن

مثال:

سه تا دستور داریم ارایه‌ی بعد تقسیم، بعد جمع و بعد تفریق

وابستگی که اینجا هم دارند به ورودی جمع خروجی تقسیم است (F_0). دستور تفریق هم رجیستر مشترک با

جمع داریم که این وابستگی اجباری که رجیستر دستور sub وابسته به کسی نیست می‌تونیم جابجایی اجرا کنیم

اینطوری out of order completion, out of order execution داشتیم

Scanned by CamScanner

اسلاید ۸۶
مثال ۸

Year:..... Month:..... Day:.....

اول یک دستور تقسیم داریم بعد ضرب و بعد هم جمع

در اینجا دستور ضرب وابسته به تقسیم است. در دستور جمع هم خروجی ورودی دستور ضرب است (F_0)

و تا زمانی که دستور ضرب F_0 را نخوانده دستور جمع اجازه ی نوشتن در F_0 را ندارد

دستور تقسیم و جمع هر دو می خوان در F_0 بنویسن که می شه output dependence که ضرب هازارد WAW

می شه پس حتی اگر دستور ضرب هم وجود نداشت باید جمع بعد از تقسیم انجام می شن

وابستگی بین ضرب و جمع هم Anti dependence است و ضرب هازارد AR ایجاد می کند

تنها راه این که دستور جمع را بنویسیم قبل از ضرب و تقسیم انجام بدیم اینکه Register Renaming کنیم یعنی اسم

رجیستر ها عوض بشه ~~[illegible]~~ که اگر hardware به صورت سخت افزاری در Run time

بتواند Register Renaming کند مسئله حل می شود

اسلاید ۸۷

مثالی از Register Renaming:

دستور اولی تقسیم می کنه، دستور دوم جمع می کنه بعد خروجی دستور دوم که F6 هست store می شه و می ره

در حافظه نوشته بشه بعد تقریبا انجام می شه بعد هم ضرب

وابستگی های موجود: در جمع F_8 داریم که ورودی است و در تقریبا اون F_8 به عنوان خروجی استفاده می شه

و در تقریبا F_{14} و F_{10} هیچ وابستگی از قبل ندارن و باید تقریبا منتظر باشه که F_8 توسط add خوانده

شه بعد بره تقریبا بنویسه

وابستگی بین add و mul است که هر دو میخوان در F6 بنویسن علاوه بر اون دستور store

ورودیش F6 است که این F6 خروجی mul است

اگر اینجا اسم رجیستر ها را عوض کنیم مسئله حل می شه

اسلاید ۸۸

موقعی که پیاده شده مثال اسلاید قبل که اومده اسم رجیستر ها رو عوض کرده تا وابستگی ازبین برده

دستوراتی که هازارد WAW و WAR دارن می شه با Register Renaming دورشون زد و اجرا شون

Scanned by CamScanner

خارج از ترتیب انجام داد. دستوراتی که متناظر RAW دارن هیچ اهمیتی نمی‌شه کرد و باید به همون ترتیب که هستن انجام بشن

Year:..... Month:..... Day:.....

اسلاید 29

* در واقعیت چه روش هایی وجود دارد که مسئله‌ی Dynamic scheduling را برای ما محقق کند و به ما امکان خارج از ترتیب بده

یکی از قدیمی‌ترین روش‌ها scoreboarding است. روشی است که یک سری اطلاعات را ذخیره می‌کند از مجموعه دستوراتی که live هستند (دستوراتی که اومدن وسط پایپلاین هستند). دستورات را نگه‌داری می‌کند و بر مبنای آن تشخیص میده که دستوری که الان درخواست ورود داره را بفرستم یا نگه‌دارم.

در اینجا اسم واحد decode را issue stage می‌گذارند این واحد می‌تواند دستورات را stall کند یا دستوری را bypass کند را اجازه دهد یک دستوری پشت دستور دیگه بره جلو و خارج از ترتیب انجام بشه

پس این واحد stage امکان اجرای خارج از ترتیب را می‌دهد و dynamic scheduling توسط این واحد stage انجام می‌شه به این معنی که واحد Fetch انجام می‌شه و ترتیب انجام را نگه می‌دارد اگر یکی نتونست انجام بشه بقیه‌ی ها اگر وابستگی ندارند می‌تونن انجام بشن

scoreboarding یک واحد مرکزی است که واحد stage را مدیریت می‌کند هر دستوری که Resource کافی داشته باشد می‌فرستد تا اجرا شود (وقتی Resource کافی دارند یعنی operand هاش آماده است و وابستگی به قبلی هاش ندارد) چیزی که هست باید حواسمون باشه اینکه data dependence وجود نداشته باشد (نه output dependence نه Anti dependence)

پس این scoreboard در issue همه چیز را چک می‌کند و هر چیزی آماده رفتن بود می‌رسه جلو و همه‌ی هزارد ها توسط scoreboard چک می‌شه تا دستوری که صلاحیت جلو رفتن ندارد جلو نرود.

جمع‌بندی اینجا با این فرض است که واحد exe یعنی ALU چندین Functional unit دارد به اضافه این که Functional unit می‌تواند پایپلاین باشد

اسلاید 10:

* scoreboard که در issue است این هست که باید تشخیص دهد یک دستور که منتقل کرده و operand هاش آماده نبود یعنی وابستگی به دستورات قبلی است، scoreboard باید حواسش باشه که این دستور که منتقل است کی operand هاش آزاد می‌شه که بره و اجرایش را انجام بده، پس باید مانیتور کند

Scanned by CamScanner

تمام آنهایی که به مشکلات جلو باشند

Year:..... Month:..... Day:.....

scoreboard باید حواسش باشد که کی یک نفر تموم می‌شه همین که کی یک نفر رجیستری را آپدیت می‌کند

در آپدیت کردن رجیستر این خواصیت که هر کسی کارش تموم شد برود همه بلندگو و Result خودش را بفرستد و باید از داخل scoreboard (همون issue) اجازه بگیره که من دیتام آماده شد آیا اجازه دارم دیتام را آپدیت کنم آن وقت scoreboard باید نگاه کند آیا هازاردی وجود دارد یا نه مثلاً دو تا دستور داریم اولی ضرب و دومی جمع است، هر دوی این‌ها با هم همزمان کارشون تموم شده اما چون دستور ضرب بالای دستور جمع بوده در برنامه دستور جمع باید منتظر بمونه تا دستور ضرب کارش را کامل انجام بده و رجیسترهاش آپدیت بکند بعد دستور جمع اجازه داره دیتاش را در رجیستر بنویسه

اوش scoreboard یک روش اجرای خارج از ترتیب هست اما out of order complation نیست چون به دستوراتی که کارش تموم شده بگه اجازه ترخیص بده

اسلاید ۱۱:

* یه دستورجیه‌بام‌هایی را می‌بیاید تا در store boarding اجرا شود:

بعد از این که instruction ها Fetch شد (instruction ها به ترتیب Fetch می‌شن) (هر جایی هم که branch شد می‌ره به اونجا و می‌ره اون instruction را می‌آره) بعد دستورات به ترتیب وارد واحد issue می‌شن. واحد issue دو تا چیز را باید چک بکند یکی structural hazard و یکی WAW.

structural hazard یعنی مثلاً یک دستور ضربی آمده و operand هاش آماده است آیا بخش اجازه می‌ده برود جلو؟ اول باید نگاه کند آیا ضرب کننده اش آزاد است یا در کار خالی است.

از این طرف scoreboard میاد WAW را هم چک می‌کند و اگر دو تا دستور داریم که issue داشته باشیم که destination یکسانی داشته باشند به دومی اجازه نمی‌ده که وارد بشه یعنی اگر دستوری رفته جلو و بعد دستوری آمده بعد من می‌خوام مقصدم همون جایی باشه که قبلی رفته اینجا issue اجازه نمی‌ده که بره جلو

حالا اگر structural hazard وجود نداشت و Functional units بیکار بود و آماده پذیرش دستور بود دستور را می‌فرستد جلو به شرطی که WAW وجود نداشته باشد. حالا ممکنه دستور بره جلو ولی ممکنه

Scanned by CamScanner

operand هاش آماده نباشد، واحد issue می‌تونه مشکلی نداره می‌تونه بره جلو و دم در function unit
که ماهاش کارداری حضور خاصتر ایمون) پس بعد از واحد issue یک واحد

Year:..... Month:..... Day:.....

Read operand داریم که بین issue و Functional unit است و اینجا هزارد های RAW ها
چک می‌شوند. برای این ممکنه operand یک دستور آماده نباشه که operand ورودی یکی همون operand
خروجی یکی دیگه‌ای است که هنوز اجراش تموم نشده پس scoreboard می‌گه این دستور نره جلو تا
واحد issue بعدی نشه که دستورات بیشتر هم پشتش نقل بشه و باعث اشغال کنه پس می‌ره و در
Functional unit منتظر می‌مونه تا operand آن از راه برسه بعد اجرا بشه
(هزارد RAW در issue بررسی می‌شه)

پس scoreboard باید مانیتور کنه که کی operand های ورودی آماده می‌شن تا دستور را اجرا کند.

بعد از اینجا فاز اجرای دستور است که هر وقت operand ها آماده شد و آماده شدن آنها از طرف scoreboard
به Functional unit اطلاع داده می‌شه دستور شروع به اجرا می‌کنه و Result آن تولید شود
گام آخر اینه که دیتایی که تولید شده write بشه که اینجا باید هزارد WAR چک شود

در نتیجه در issue هزارد WAW چک می‌کنه، هزارد RAW در ورودی Functional unit
چک می‌شه و هزارد WAR در خروجی Functional unit چک می‌شه و همه‌ی اینها را scoreboard
مانیتور می‌کنه که اتفاق بدی نیفته

اسلاید ۱۲:

شکل سیستم scoreboarding

اسلاید ۱۳:

روش scoreboarding یک روش single issue است، single issue در مقابل
multiple issue قرار دارد

single issue یعنی در هر کلاک یک دستور می‌ره جلو، در مقابل multiple issue است
یعنی واحد issue هر کلاک n تا دستور را می‌فرسته جلو پس دستورات sequential جلو نمیرن و
n تا n تا می‌رن جلو

Scanned by CamScanner

وقتی یک دستوری issue می‌شه به سمت واحد Functional unit این Fu دیگه بلاک می‌شه

Year:..... Month:..... Day: تا زمانی که این instruction اجرایش شروع نشه یا اصطلاحاً dispatch

نشود یعنی واحد Read operand که دستورات را می‌فرستاد جلو می‌کند dispatch کرده

یعنی به جلو فرستاده و به دست تصرف کننده رسیده

اسلاید ۳۱۴

این score board که دیدیم اگرچه چیز خوبی است اما یکسری محدودیت‌ها هم دارند

روش اصلی scoreboard روش آقای Tomasulo است

* Tomasulo می‌شه که چک کن ببین کی operand ورودی دستور آماده است و دستور را فرستاد جلو

و یک قابلیت نسبت به روش scoreboarding داره که باعث می‌شه تعداد دستورات کمتر بشه

Tomasulo می‌شه بیایم Register Renaming را هم اضافه کنیم به سخت‌افزار که دستورات را انجام

WAW و WAR ها را از بین ببرد و اینجوری سرعتش بالاتر برده پس Tomasulo با

Register Renaming این دو تا حذف را انجام می‌کند و out of order execution را تبدیل می‌کند به

out of order complition

تفاوت دیگه بین Tomasulo و scoreboarding این که در scoreboarding یک واحد scoreboard

مرکزی داشتیم در حالی که در Tomasulo واحد مدیریتی پخش شده و اطلاعاتی که باید ذخیره کنیم در stage های

مختلف پایپلاین پخش است به صورت distributed

Tomasulo یک واحد سخت‌افزاری به سیستم اضافه می‌کند به نام reservation station (Rs)

که حاوی موارد زیر است:

۱) Rs را جلوی هر Functional unit می‌ذاریم که این Rs یک صف است که فقط این که

operand ها آماده هست یا نیست را خلش نیست را اطلاعات دیگه هم را خلش هست. این Rs هر

خانه‌اش خود instruction را خلش هست به اضافه operand های این instruction اگر

وجود دارند. Tomasulo می‌شه اگر یک دستور دو تا operand داره و یکی از operand هاش فقط

آماده هست، اونی که آماده هست را بخوان و همان جا منتظر بمان دم در Fu کنار این instruction

می‌ایستد و منتظر می‌موند. چه نیازی داره یک operand را الان بخوانیم؟ چون ممکنه یک دستور بعد از

Scanned by CamScanner

این دستور باشه که بخواد از اون operand استفاده کنه و آن ما آپدیت نکنه اینجوری دیگه کسی مانعش نمی‌شه.

Year: Month: Day:

پس operand هایی که موجود هست را buffer می‌کنه در این صورتی انتظار

به اضافه این که برای اون operand که هنوز آماده نیست این Rs مشخص می‌کنه که چه کسی قرار است به من دیتا بدهد یعنی دوتا operand داریم اگر هیچکدوم آماده نیست پس Rs میاد مشخص می‌کنه که operand ورودی من را چه کسی قرار است تولید کند، اگر هم وجود دارد که می‌نویسش

اسلاید ۵۱۵

کاری که Tumasulo می‌کنه اینه که Rs هر وقت که operand هر دستوری بود آن را بلافاصله می‌خونه و منتظر اون کسی نمی‌شه. instruction هایی که pending هستند ~~مشخص می‌کنه~~ (یعنی یکی از operand های ورودیشون آماده نیست) مشخص می‌کند که operand ورودی من را چه کسی تولید می‌کند یعنی operand ورودی من توسط کدام Functional unit قرار است تولید شود و چون هر Fu یک صف n تایی از دستوراتش وجود دارد یکی از اون n تا دستور ~~[illegible]~~ (یا یکی یا هیچکدوم) از آن Fu داره استفاده می‌کند، از اون n تا دستور چون Fu هر کدوم operand هاشون آماده نشده می‌تونه بره جلو و اجرا بشه، تا زمانی که اون دستور از Rs یک Fu خونده می‌شه و می‌ره اجرا می‌شه تا زمانی که اون کارش کامل تموم نشه اطلاعاتش از صف Rs خالی نمی‌شه. حالا یک دستور دیگه چون یک Fu دیگه منتظر یک دستور دیگری است و اون دستور دیگر قطعاً توی یکی از Fu هایی است که Rs به اسم خودش داره پس هر دستوری که operand اش آماده نیست می‌ره توسط کدام Fu و کدوم ردیف در صف انتظار پس قرار است تولید شود

* هر کسی که دیتاش آماده می‌شه یعنی هر دستوری که تموم می‌شه و هر Fu یک دستور را کامل اجرا می‌کنه و دیتا آماده می‌شه اون دیتا broadcast می‌شه روی یک bus که بهش می‌گن common data bus (CDB)

این CDB ~~[illegible]~~ باسی (Bus) هست که دیتایی که روش قرار گرفته می‌شود هم از طریق آن به جستجو می‌رود که چه کسی را آپدیت کند و هم این خبر را به تمام Reservation station ها مخابره می‌کند یعنی ~~[illegible]~~ همه‌ی Rs ها هویت در حال ما نیو کردن این bus هستند چون در Rs دستوراتی وجود دارد که این دستورات operand هاشون هنوز آماده نیست که اون operand ها وقتی آماده شد روی این bus قرار می‌گیره

Tomasulo می‌گه کسی که operand اش آماده نیست اگر در scoreboard باید صبر می‌کرد که operand

Scanned by CamScanner

توسط یک دستور آماده بشه و در یک رجیستر فایل نوشته بشه و بعد از رجیستر فایل بخونیم من نیازی نمی‌بینم که شما
Year:..... Month:..... Day:.....
صبر کنی تا این دیتا در رجیستر نوشته بشه و بعد بیای از رجیستر بخونی

خودمان با آپدیت شدن رجیستر دیتا بیاد روی bus و من از روی bus بردارم دیتا رو
در لحظه دیتایی که روی Bus قرار می‌گیرد گفته می‌شود این دیتا را چه کسی تولید کرده است و از کدام FU آمده و در اونجا
کدام صندلی انتظار بهش داده شده بوده

در روش Tomasulo از n تا دستور که در RS هستند می‌تونند اگر output dependency وجود دارد
اگر WAW hazard وجود دارد هازارد WAW را می‌شه رفع کرد به این صورت که فقط نفر آخر از این n نفر که
هازارد را ایجاد کرده اند رجیستر را آپدیت کند و بقیه‌ی هازرجیستر را آپدیت می‌کنند یعنی اگر دو تا دستور در یک
رجیستر می‌خوان بنویسن فقط نفر دوم اجازه داره آپدیت کنه و نفر اول اجازه‌ی آپدیت را نداره حتی پس
نفر اول برای چی داره اجرا می‌شه و دیتاش را چیکار کنه؟ قطعاً اگر دستوری وجود داره که دیتایی تولید می‌کنه و در یک
رجیستر ست شده که نوشته بشه و دو یا سه تا instruction بعدش یک دستوری هست توی همون رجیستر بنویس
این دفعه‌ش اینه که life time دیتایی که دستور اول تولید می‌کنه در حد دو سه تا instruction بعد از خودش
هست و تا زمانی که instruction بعدی آن رجیستر را آپدیت کند پس قطعاً دو سه تا دستور بعد از
دستور اول هستند که منتظر خروجی دستور اول در RS هستند پس حالا که منتظر هست دیتا را از روی
CDB می‌خونه؛ اونی دستوری که منتظر تولید یک دیتا هست می‌ره آن دیتا را از رجیستر فایل نخونه
اگر تولید آن دستور توسط instruction قبلی در حال اجرا بوده، پس می‌ره و از روی bus می‌خونه و چون
از روی bus می‌خونه نیازی نیست نگران این باشیم که دستور اول رجیستر فایل را آپدیت نکرده، دستور دوم
آپدیت کند نهایت می‌کنه به همین راحتی WAW hazard رفع می‌شود

یک نکته دیگه از چیزی که الان توضیح دادم: وقتی یک instruction، operand در روش ما رجیستر
RS هست و RS را می‌ره یکی instruction هایی که در حال اجرا است تولید کنه وقتی آن instruction
را در RS نگه می‌داریم و بهش می‌گیم منتظر بمون تا دیتا را برات تولید کنم اونجا بهش می‌گن که operand
ورودی را از RS بیار؛ بهش می‌گن operand ورودی را از Reservation station functional unit
فلان رو بیس تولید می‌کنه، پس می‌خوادبری از RS بخونی چون در RS هنوز دیتای ما
آماده نیست و وقتی آماده شد قبل از این که توی RS نوشته بشه روی bus برای ما ارسال می‌شه
پس جای این که خودمون به آپدیت شدن رجیستر فایل با شه خودمون به آن instruction بابته
که می‌خواد رجیستر فایل را آپدیت کند و عملاً در روش Tomasulo ما جلوی هر FU چندین

Scanned by CamScanner

ضد ریسک انتظار داریم و هر ضد ریسک انتظار چندین فیلد داره که این فیلد ها حداقل به اندازه ی دیتا رجیستر operand

Year:..... Month:..... Day:.....

علاوه بر اضافه شدن این که به اندازه ی دیتا آدرس Rs های فرستنده

علاوه که یکی می خواد operand من را تولید کند به اضافه این که به اندازه ی خود این instruction که چیست

باید اطلاعات ذخیره کنیم. مجموع اینها باید از کل رجیستر هایی که بردارنده به عنوان رجیستر های visibale بیشتر شود.

مدل بردارنده ها واحد load و store هم داریم (دستوراتی که دیتا را می خوانن و می نویسن) اینها هم برای

خودشون یک ساختار Resarvation station دارند. دستورات load ها هم آدرس هاشون توسط یک مدی

دستورات Computation ایجاد تولید شود. دستورات store هم operand هاشون از عملیات

Computation تولید می شود اینها هم باید همین کاری که بقیه می کنند برای خودشون داشته باشند یعنی واحد

load و store هم Rs برای خودشون دارند و همانطور می کنند اتفاقاتی که برایشون می افته و

دستورات بعدی ها را handle می کند

اسلاید ۱۶

ساختار کلی Tomasulo 8

یک instruction fetch unit داره که instruction ها را Fetch می کند یک instruction queue

داریم که یک صفی از instruction هاست که واحد Fetch بدون نگاه کردن این که چه چیزی توش هست

در هر کلاک هر یک instruction میاره ولی زمانی که این صف جای نداره اجرای من و جلوی رن یا

تا جایی که صف جا داره instruction ها را می آورد

هر دستوری که اومد در صف بسته به این که از کدام دستور چیست (مثلا یک ضرب که جمع کننده داریم اینطوری

حکم Rs دارند) می رود Rs یعنی ضرب، جمع یا load و store هست داخل Rs می ره و کسی

جلوی دستورات را نمی گیرد (چون هر Fu برای خودش یک صف داره دستورات می رن اونجا می نشین).

Tomasulo تنها اجرای خارج از ترتیب داره که Complition خارج از ترتیب هم داره

اسلاید ۱۷ 8

* پس دستورات بعد از Fetch شدن و در صف قرار گرفتن سه کام دارند:

کام issue در اینجا دستورات چک می شن بین Rs های مختص به Fu خودشون به

شرط این که اون Rs جا داشته باشد. هر دستوری که روانه شد به سمت Rs، stall می شه تا operand هاش

آماده بشه اگر Rs هم خالی نباشد قبول است در جلوی مهم منتظر بمونه

Scanned by CamScanner

۲) دستور اجرایی شب محض این که Rs به operand حالت آماده شد

Year:..... Month:..... Day:.....

دستورات load و store ~~[illegible]~~ به این دلیل درست

دستورات computation می‌شوند که می‌خوان آدرسشون تولید بشه به اضافه این که دستور store ، اگر بزنی

~~[illegible]~~ store که از یک عملیات computational باید بیاد

Tomasulo یک محدودیت دارد و اینه که می‌گه چون من فقط حواسم به data dependency هست و چون لازم

دستورات را خارج از ترتیب complite می‌کنم و به دستوری که اجرا شد اجازه می‌دهم که اگر اجرا شد

دیتاش را در رجیستر آپدیت کند و خیالم راحت از این آپدیت کردن مشکلی برای کسی ایجاد نمی‌کند Tomasulo

می‌گه که اگر من اجازه می‌دهم دستور خارج از ترتیب complite شود حواسم اینه که حتماً نشم این دستور

باید اجرایی شد ولی دستور branch می‌گه که اگر branch prediction اجرا کردی و ~~[illegible]~~

حدسی یک مسیر را جلو بردی غلط بود branch که اشتباه زدی آن وقت دستوری که بعد prediction

زمت جلو آمد که دیتاش را write کند در لقصد و رجیستر را آپدیت کند ناچار پیش میاد چون نباید اجرایی شد

پس Tomasulo می‌گه همین که به دستور branch رسیدیم باید همه‌ی این اجرای خارج از ترتیب‌ها را جمع

بزنیم و صبر کنیم که تکلیف branch مشخص بشه

این کار خوب نیست چون دست Tomasulo را می‌بندد چقدر چه چیزی بستگی داره به برداری دستورات

branch و اهمیت branch prediction (دستورات branch چه‌اش و انشون زیاد است)

۳) مشکل write result :

دیتاش را روی bus می‌نویسه و هر کسی این دیتا هم به دست رجیستر فایل می‌رسه هم کسانی که منتظر این

دیتا بودن

اسلاید ۸۱

مثال از Tomasulo:

فرض کنید ~~[illegible]~~ بالای این اسلاید را در نظر بگیریم ، فرض کنیم همه‌ی این دستورات issue شدند یعنی همه‌ی

این ۶ تا دستور Fetch شدند از این ۶ تا هم خوشه شدند و پخش شدند فقط F۲ ها و در Rs ها

قرار گرفتند ، دو تا دستور اول Execute هم شدند و دستور اول write هم کرده و رجیسترش را آپدیت کرده ،

دستور ضرب و ~~[illegible]~~ منتظر برای این جلو نمی‌رفت که F_2 می‌خواد که به دستور دوم وابسته هست ، دستور

تقسیم F_6 می‌خواد که الان توسط دستور اول تولید شد اما F_4 می‌خواد که به ضرب وابسته است

Scanned by CamScanner

دستور add هم F_8 می‌خواد که sub باید تولید کند و F_2 که دستور دوم باید تولید کند پس این چهار تا دستور وابستگی RAW دارند

Year:..... Month:..... Day:.....

به فرض این که همه‌ی ایفاهای Rs هستند جدول دوم وضعیت این دستورات در Rs را نشان می‌دهد

واحد load/store، Rs دو تا ردیف داره که هر ردیف یک سری فیلد داره که یکیش می‌گه busy هست یا نه بعدی می‌گه دستور چیه هست و operand چیه، دو تا قسمت بعدی این است که operand های ورودی من چیه هستند، operand ورودی اول یا دوم که آماده بود از رجیستر خوانده می‌شه و اینجا قرار می‌گیره

دو تا فیلد دیگه داره که اگر از این operand ها آماده نباشه مشخص می‌کنه چرا آماده نیستند و منتظر چیه یعنی هست که آماده بشه یعنی Q_j و Q_k مثلا Rs هستند که قرار است operand ها را تولید کنند، این A هم آدرسی هست برای دستورات store/load توضیحات این در دقیقه ۱۹ هست

اسلاید ۱۹:

یک مثال دیگه:

Tomasulo این ۵ تا دستور را با هم می‌ریزه در صف و بعد باید این دستورات با این Fu ها چیش بشه، فرض را بر این می‌زاریم که Tomasulo سه دستور اول را خوانده و می‌رسه به branch prediction branch می‌شه برگرد فرض کنیم اون تیکه‌ای که نشستیم در Tomasulo به دستور branch رسیدی متوقف شو را برداشتیم و چیش می‌کنیم باید Iteration بعدی Loop را اجرا کنی پس Tomasulo همیشه از اول سه تا دستور اول را Fetch می‌کند و می‌ریزه در صف و اونها را می‌توفیزیست به سمت Rs ها

اسلاید ۲۰:

حالا اگر طبق جدول این اسلاید سخت‌افزار دوتا Iteration از این loop را Fetch کرده باشه یعنی دو دور instruction ها را آورده باشه و در صف ریخته باشه، می‌خوام ببینم وضعیت Rs ها چطوره است

توضیحات این قسمت دقیقه ۲۵

اسلاید ۲۱:

جمع‌بندی Tomasulo و scoreboard:

- هر دو inorder issue هستند یعنی به ترتیب دستورات را Fetch می‌کنند و در نهایت واحد issue و issue اگر بخواهد توسط می‌تونه بره جلو

- در scoreboard کل مجموعه‌ها stall می‌شود اگر وابستگی RAW وجود داشته باشد اگر وابستگی RAW وجود داشته باشد در فرمول واحد scoreboard stall می‌شود

Scanned by CamScanner

- در Tomasulo، operand forwarding داریم ولی در score board نداریم (چون که operand روی bus نوشته می‌شود)

Year:..... Month:..... Day:.....

- Tomasulo، hardware complexity بیشتر از scoreboard دارد

* روش Tomasulo هم تضمینی برای control hazard ندارد و فقط برای data hazard است

ویدیو نهم (Lec5-chap3-E)

اسلاید ۴:

* hardware-based speculation (امری speculative برمبنای سخت‌افزار):

speculation یعنی حدس زدن یعنی چیزی را اجرا کنیم که از اجرای آن مطمئن نیستیم.

speculation می‌شه ما با دستورات branch جلو بیاریم، وقتی به یک دستوری می‌رسیم که مطمئن نیستیم بعدش چیکار کنیم آیا مثل Tomasulo عمل کنیم یا با تک و تردید جلو بریم؟

speculation می‌شه با تک و تردید جلو برید ولی حواستون باشه کاری که انجام میدید کار مطمئنی نیست، پس این روش راهی را نشون میده که با حدس و گمان جلو بریم ولی منطقی برنامه بهم نریزه

نکته: ما سه نوع outup order داریم ← ۱) outup order execution ۲) outup order complition

۳) outup order issue

روش Tomasulo و scoreboard هم outuporder execution و هم outuporder complition دارند اما هر دوی اینها inorder issue هستند یعنی دستورات را به ترتیب issue می‌کنند

حالا می‌رسیم سراغ speculation:

با توجه به این که قبلا گفتیم فراوانی دستورات branch خیلی زیاده و basic block های ما طولشون کوتاه هست و ما زود به زود به دستورات branch می‌رسیم در روش اجرای speculative می‌رسیم که وقتی به دستور branch رسیدیم توقف نکنیم و بر اساس مسیری که حدس می‌زنیم درست باشه پیش بینی کردیم جلو می‌رویم اما با توجه به این که دستوراتی که بعد از branch دارن اجرا می‌شن دستوراتی هستند که از اجراشون مطمئن نیستیم یک خروجیه باید بذاریم سر اجرای speculative ها و این خروجی این است که ما commitment یا اتمام کامل دستور را انجام ندیم تا زمانی که از راهی که رفتیم مطمئن باشیم

به صورت کلی speculation می‌شه هر تریکی که دلت می‌خواد دستورات را issue کن و اجرا کن و بذار Result را تولید کنن اما آن آخر همه را مرتب کن و کاری کن که هر آن که با حدس و گمان اجرا شده بخش اجازه نده چیزی را ایجاد کنه مگر این که حدسش تکلیفش روشن شده باشه

Scanned by CamScanner

پس commit شدن یک instruction یعنی این که به این instruction اجازه بدهیم که رجیستر ش را آپدیت کند

و output را در نهایت بنویسد و کار ش تموم شود

Year: Month: Date:

Subject:

برای این کارها ما نیاز داریم یک سخت افزاری را اضافه کنیم به سیستم به یک صف انتظار یا queue خروجی.

(در Tomasulo ما یک صف ورودی داشتیم که RS بود) که این جلوگیری می کند از این که یک action غیرقابل

ارتکاب شدن اتفاق بیفتد

یعنی action غیرقابل ارتکاب شدن یعنی دستوری رجیستری را آپدیت کند عامل دستوری رجیستری را آپدیت کرد

دیگه نمی توانیم اون رجیستر را به value قبلش برگردونیم اگر نه مجبوریم کارهای کار اشتباهی است پس تا زمانی که تضمین نشدیم

که باید رجیستر آپدیت شود باید آن کسی که value رجیستر را تولید کرده معطل نگه داریم تا دستور branch که قبل از

اون بوده تعیین تکلیف شود و بعد بحثی را اجازه بدهیم که رجیستر را آپدیت کند یا آن را دور بریزد

اسلاید ۵۵

* برای این واحد تخصیص مفهوم Reorder buffer مطرح می شه

Reorder buffer، result هایی که instruction ها تولید می کنند را در خودش نگه می دارد پس یک بافر است بین

complite شدن و اتمام اجرای یک دستور با commit شدن دستور

هر فیلد reorder buffer چهار قسمت داره:

۱) این entry متعلق به چه instruction هستند (چه است، تویی است، load، store است، branch است)

۲) این instruction قصدش کجاست یعنی این دیتایی که توی بافر ذخیره می کنیم و از خروجی Alu آمده (از خروجی

Functional unit اومده مراقبه نیابه و در چه رجیستری نوشته شده)

۳) value آن چیه است یعنی چه چیزی قراره در فلان رجیستر توسط این دستور نوشته شه

۴) این ردیف آزاد است یا توسط یک instruction اشغال شده یعنی Ready filed است یعنی آیا

instruction که می خواد اینجا بنویسه تموم شده یا نه

اینجا اجازه می گیرند صبر کنیم تا instruction تموم شده پس در reorder buffer چطوری می

جلوتر جواب این سوال را می دهیم

SOBHAN

Scanned by CamScanner

ما با وجود reorder buffer باید Reservation station که توی Tomasulo بود را دستکاری کنیم 8

در Tomasulo یک instruction که وارد Rs یک Fu مثل جمع کننده یا ضرب کننده شد چند تا فیلد کاشت

Subject: Year: Month: Date:

این instruction چه هست، operand های ورودیش چه هستند و اون operand هایی که available بودن

آنجا نوشته می‌شدند و رزرو می‌شدند و اگر operand ها کامل بود instruction می‌رفت اجرا می‌شد و اگر

operand ها کامل نبودند منتظر می‌شد چرا operand وجود ندارد و اگر آپدیتی available نیست منتظر

می‌شد که operand ما توسط دستوری اجرا می‌شود که آن دستور در Rs یکی از Fu هاست و در Rs

این instruction که الان داریم می‌نویسیم که اون operand کجا در کدوم ردیف ... است و حواست

باشد هر وقت خروجی که روی bus هست همونی بود که تو دنبالش هستی آن دیتا را از روی bus

بردار و در Rs قرار بده تا مجموعه ورودی‌هات کامل بشه و اجرا را انجام بدی

در spaculation اگر دستوری operand های ورودیش آماده نباشد، می‌رویم توی Rs

آن operand ورودی را تولید می‌کند شناسه می‌گیریم operand ورودی در کدوم ردیف Reorder buffer است

نکته: تفاوت Rs و Reorder buffer ← Rs قبل از Fu قرار می‌گیره و Reorder buffer بعد از Fu،

جایگاه هر Fu، Rs داریم و این ردیف‌ها ربطی به هم ندارن و مستقل از هم عمل می‌کنند، اما ما یک

Reorder buffer کلا بیشتر نداریم و جایگاه Reorder buffer، یک global و buffer داریم که

فایده‌اش این که دیتا از کدوم Fu و از کدوم Rs خارج شدند می‌رن در این بافر چون نقش بافر این است

که فایده‌اش این که این instruction ها باید ترتیب وارد چه Fu شده باشند در نهایت همشون

به صورت ترتیبی commit می‌شن که این منطق برنامه است

یک نکته مهم در مورد spaculative execution اینه که وقتی که instruction را می‌خواهیم issue

کنیم یعنی instruction را Fetche کردیم و خواستیم بفرستیم به سمت Functional unit اولا باید

در اون Fu در Reservation station وجود داشته باشه (شرطی که برای Tomasulo هم

وجود داشت) اما یه کار دیگه هم باید انجام بدیم و اون اینه که instruction قبل از این که به سمت Fu

ارسال شود خودمان با تخصیص یک ردیف از Rs باید یک ردیف از reorder buffer هم

همین الان باید بهش اختصاص بدیم چون اون اگر تو نتیجه فهمیدیم که باید ترتیبی وارد شده از

instruction که میره جلو اول اگر هم یک فضایی برایش رزرو کردیم)

اسلاید 86

SOBHAN

* در مورد reorder buffer :

- در مرحله issue کردن instruction ها یک Rs و یک reorder buffer را اختصاص می‌دهیم به دستور

ردیف یک ردیف

۹۸

و اگر هر کدوم از اینها وجود نداشته باشند دستور می‌تونه جلو بره (یعنی Rs جا داشته باشد ولی ROB نباشد پس
من نداریم دستور را جلو نمی‌ریزیم). اگر تونستیم هر دو را تخصیص بدیم دستور ما وارد Rs می‌شه و از operand های
Year: Month: Day:
که آماده هست را می‌خونه و اونهایی که آماده نیست را منتظر می‌مونه تا آماده بشه

- زمانی که operand ها آماده شدند شرط آن است که آن دستور اجرا شود اگر آن FU در حال اجرای
دستور دیگری نیست

- دیتا از FU تولید شد حالا باید دیتا را به همراه تگ reorder buffer را روی comand data bus
می‌نویسیم

تگ ROB ← در Tomasulo می‌گفتیم این دیتا را کدوم Rs تولید کرده ولی در اینجا اینو می‌گیم بهش
می‌گیم این دیتا را کدام خونه (مدخل) از ROB تولید کرده پس در Rs هم از instruction هایی که
operand هاشون آماده نبوده تا الان و منتظر Bus گوش می‌کنند مشخص نیست که ببینند دیتا را کدوم
Rs تولید می‌کند بلکه گوش میدن ببینن دیتا را به کدوم ردیف ROB برود (می‌دونیم دیتا از خانه‌ای که
رزرو شده اون آخر بهش ردیف ROB اختصاص داده شده)

- حالا که دیتا در Rs ذخیره شد میره سراغ commit و می‌گه دیتایی که از FU خارج شده و در ROB
بالاتر از اینه باید منتظر بماند که نوشته شه و value را در رجیسترها آپدیت کند

حالا اتفاقی که می‌افته اینه که ما جلو اینا نیاز به ROB داریم؟ به خاطر این که دستورات پیش و پیش اجرا
شدن و این پیش و پیش اجرا شدن شامل دستورات branch هم می‌شود. حالا دستوری خودش را
ROB ما می‌رسوند به دیتایش را نوشته و مشخص است که نوشته شده که نوشته شده اگر این دستور در منطق
برنامه جایش یک دستور branch باشد آن وقت در ROB بالاتر از این دستور یک دستور
branch است که هنوز complite نشده و تا زمانی که آن دستور branch complit نشده نوبت
این دستور نمی‌شه. جلو دستور branch complite می‌شود و branch می‌دونیم پیش بینی بزنیم ولی
شما با زدن پریدن دستورات بعدی را اجرا کردی و تمام RoB entry های که بعد از entry
branch هستند همشون از مسیری اجرا شدند که نباید اجرا می‌شدند پس نوبت branch شد و
نپریدید حالا باید تمام دستورات بعدی را دور بریزیم. این دور ریختن مشکل برنامه را عوض نمی‌کند و به جایی
بدی می‌خورد چون ما چیزی را آپدیت نکردیم و در نهایت تمام دستوراتی که منتظر هستیم آپدیت می‌شن

Scanned by CamScanner

پس هیچ رجیستری تا قبل از این که instruction commit شود آپدیت نمی‌شود و روی misprediction هر آنچه که داریم را دور می‌ریزیم

Year:..... Month:..... Day:.....

- Exception (استثنا):

می‌دونیم که در روند اجرای دستورات در پایپلاین برداشته ممکن Exception رخ بده که بدهی‌ترین آن تقسیم بر صفر است. وقتی یک Exception رخ میده یک Exception handling باید اجرا بشه. وقتی که ما دستورات را به صورت out off order اجرا می‌کنیم یعنی الان ممکنه دستوری اجرا بشه که در منطق برنامه بعد از دستوری می‌آید که هنوز اجرا نشده و یکی از دستورات ممکن است منجر به exception شود. یک دستور خاص که می‌تواند منجر به exception شود وارد پایپلاین می‌شود و exception آن رخ نمی‌دهد مثلاً یک تقسیم داریم یعنی یک تقسیم بر صفر انجام می‌شود و وقتی این exception رخ می‌دهد آن exception handeling را انجام می‌دهد و بعدش برمی‌گرده به برنامه و کارش را ادامه می‌دهد. در لحظه‌ای که exception رخ می‌دهد باید تمام این instructionهایی که قبل از آن instruction که منجر به exception شده است، کارشون را انجام داده باشند و هیچ instruction که در منطق برنامه بعد از instruction که منجر به exception شده است نباید اجرا شده باشد. در این صورت در اجرای out of order مثل Tomasulo و scoreboard مشکلی که دارن اینه که در منطق برنامه یک دستور تقسیم داریم و یک جمع که بعد از آن تقسیم قرار گرفته و این دو تا دستور هیچ ربطی به هم ندارن. Tomasulo و scoreboarding در زمانی که تقسیم داره انجام می‌شه و ۳۰ کلاک طول می‌کشه اگر به جمع برسند، جمع را می‌فرستند سمت جمع کننده و من بر اجرا کن و result را هم بنویس و منتظر عمل تقسیم نشو. جمع انجام می‌شه و یک رجیستر را آپدیت می‌کنه و بعد در آن تقسیم کننده یک Exception رخ می‌ده. مشکلی که وجود داره اینه که زمانی که ما Exception handeling را انجام می‌دهیم نباید آن جمع انجام می‌شده و چون انجام شده سیستم در حالت unstable است.

اما آن دسته‌ای که می‌تونن رخ بده اینه که دستورات منجر به Exception می‌شه از یک دستوری جلو زده باشه و در حال اجرا باشه مثلاً یک دستور جمع داریم که بالای دستور تقسیم قرار گرفته و وابستگی هم ندارند ولی الان دستور جمع بخاطر این که واحد جمع کننده مشغول است یا وابستگی داره به چیزهای دیگه نمی‌تونه اجرا بشه به تقسیم می‌رسن پس مثلاً تو بردِ جلو واحد اشتباه، می‌ره اجرا می‌شه و exception روی آن

Scanned by CamScanner

رخ می‌دهد و در آن لحظه می‌ره سراغ exception handler و سیستم باید در وضعیتی باشد که تمام instruction های قبل از این exception انجام شده باشند و تمام رجیستر‌ها را آپدیت کرده باشند و هیچ کسی بعدش هم اجرا نشده باشه اما دستور جلوتر قبل از این که اجرا نشده پس سیستم در واقعیت به اونجایی نرسیده که باید exception رخ بده و این هم مشکل ایجاد می‌کند

آیا روش speculative execution با ROB مشکل برطرف می‌کند؟ با توجه به این که ما همه چیز را داریم آن‌ها مرتب می‌کنیم در روش ROB این قابلیت وجود دارد که می‌گه که اجازه بدیم دستورات ما در زمان Exec قبل‌شون هر اتفاقی می‌خواد براشون بیفته حتی exception و تقسیم بر صفر هم رخ بده، ما هیچ واکنشی نشون نمی‌دیم و تا زمانی که اون دستور در ROB نوبتش نشده نیست، هر آنچه که در گذشته اتفاق افتاده هیچ عکس العملی نشون نمی‌ده، بنابراین اگر یه دستور تقسیم بر صفر داریم و اون دستور تقسیم نوبتش تقسیم بر صفر رخ داده باشه و دارای یک Exception باشد exception handler را call نمی‌کنیم و به روی خودمون نمی‌آوریم اصلاً که exception رخ داده و به جوابی سوالات اختیاری نمی‌کنیم و اجازه می‌دیم دستور تقسیم بعدی و دیتا را روی common data bus بنویسه و دیتا را در ROB هم بنویسه و می‌گه بعد هر کاری دوست داری بکن و هر وقت نوبت تو شد می‌دونیم باید چیکار کنیم و ROB صبر می‌کنه تمام دستوراتی که قبل از اون دستور تقسیم هستند دیتاشون تولید بشه و بیان وارد ROB بشن و اون دیتاها را بنویسه توی رجیستر‌ها و همشون تموم شن برن بیرون از سیستم و هر دستوری که بعد از تقسیم بوده هم در ROB قرار می‌گیره و result را در ROB می‌نویسه و چیزی را آپدیت نمی‌کند تا ببینه فرق می‌کنه دیتا دستور تقسیم را نگاه می‌کنیم می‌فهمیم تقسیم بر صفر رخ داده، می‌گیم خب هر آنچه که قبل از این exception بوده را انجام دادم و هر آنچه بعد از exception است که در ROB هست و جایی نرسیده که چیزی را خراب کنن پس از این لحظه می‌ره exception handler را call می‌کند و ROB را خالی می‌کند و هر آنچه بعد از این اجرا شده را دور می‌ریزد مثل وقتی که miss prediction در branch رخ می‌دهد و به همین راحتی مشکل exception را حل می‌کند و تا زمانی که دستوری که exception نوبتش رخ داده آماده commit شدن نباشد یعنی در ROB نوبتش نشده تا نوبت instruction که مربوط به آن Exception هست را اعلام کند این یعنی آماده commit شدن) هیچ اتفاقی نمی‌افتد پس مسئله حل است

اسلاید ۸:

(از توضیح خاستگاه صفحه ۱۴)

ساختار طبق مفهوم حاکم بر اجرای ROB و speculative ← خیلی شباهتی با Tomasulo ندارد

اسلاید ۹:

سناریو از بردن reorder buffer (صفحه ۴۳) (ایجاد commit یعنی تموم شدن رسمی)

Scanned by CamScanner

همه ی این تلاش ها را می کنیم تا در نهایت پایپلاین را تا می توانیم پر نگه داریم و در نهایت CPI نزدیک به 1 برسیم
و هدف نهایی ما اینه که ما در هر کلاک بتوانیم یک instruction را فرستیم جلو و

Year:..... Month:..... Day:.....

تمام stage های کلاک پایپلاین در هر کلاک پر باشه. پس با فرض این که همه چیز عالی پیش بره و هیچ اتفاق
بدی نیفته و به بهترین نحو عمل کنه تحت performance که خواهیم داشت برابر با CPI=1 در حالی که در
عالم واقعی CPI هیچ وقت یک نمی شه چون هیچ وقت حال ایده آل که هیچ مشکلی نباشه نداریم

ویدیو ششم (Lec6- chap3-F)

اسلاید ۳۴

* multiple issue and static sch :

در یک کلاک بیش از یک instruction را تحمیل کند و تلاش می کند CPI را کمتر از یک کند و قبل
از این که کلاک بخواد تحمیل کند باید بتواند بیش از یک instruction را issue کند. از روش multiple issue
استفاده می کنیم که n تا دستور را fetch می کنیم و n تا دستور را هم زمان به سمت reservation station سوق می دهد

سه ساختار کلاسیک برای سیستم multiple issue وجود دارد:

- پردازنده های superscalar statically scheduled که به process های تقسیم می شود
که در هر کلاک multiple issue دارند و multiple instruction را سوق می دهند به سمت
واحدهای Functional. حالا برای این که هم زمان n instruction را فیش کنیم بین واحدهای
اجرایی، یک حالت این است که به صورت static بیایم schedule کنیم چند تا
instruction که می توان هم زمان اجرا شن و وابستگی به هم ندارند را پیدا کنیم و چند تا چند تا instruction
که می خوایم از قبل باید بیارهم نداشته باشه

- حالت دیگه ای که در مقابل static است حالت Dynamic است که نقش افزار ما
این superscalare بودن پردازنده را تأمین کند و نقش افزار ما، n instruction که
را هم ببیند و وابستگی ها را detect کند و بلد هیچ کار را داشته باشد و کار رو پیش کند بین واحدهای
exeption (را حالت Dynamically scheduled superscalar است)

- VLIW (very long instruction word): یک حالت خاص دیگر از پردازنده هایی که

Scanned by CamScanner

به صورت statically schedule می‌شن، پردازنده‌های VLIW ها هستند

Year:..... Month:..... Day:.....

اسلاید ۵۵

* این جدول یک جمع‌بندی از مجموعه ساختار پردازنده‌های super scalare

این جدول ۵ حالت از حالت‌های ساختار super scalare را نشان می‌دهد. حالت ۱ پردازنده‌های static است که واحد issue به صورت Dynamic عمل می‌کند، hazard آنها تحت افزاری است scheduling آنها static است یعنی از قبل علوم است که چه instruction هایی با هم قرار است برن جلو، Execution آنها inorder است و معمولاً هم در embeded processor ~~embeded~~ استفاده می‌شه مثل arm و MIPS (arm مثل Cortex A53)

حالت دیگه پردازنده‌های Dynamic هستند، issue آنها داینامیک است، هازارد به صورت تحت افزاری است و scheduling آنها به صورت Dynamic است یعنی تحت افزار تعیین می‌کنه چه instruction هایی را جلو بفرستیم، Execution آنها می‌شه پردازنده‌های out of order که speculation دارند مثلاً Tomasulo را در نظر بگیرید که به جای یک instruction، instruction ها را به حال تعویق issue کنه ولی همچین پردازنده‌ای وجود نداره چون الان همه‌ی پردازنده‌ها speculative عمل می‌کنند یعنی با توجه به این که basic block ها خیلی کوتاه است و نمی‌تونیم سر هر branch متوقف بشیم و branch prediction را نداشتیم که با دستورات branch هم کارمون را جلو ببریم تمام پردازنده‌هایی که الان داریم به صورت out off order عمل می‌کنند، speculative هم هستند پس می‌ریم سراغ حالت ۳

مورد ۳: issue به صورت داینامیک است، هازارد به صورت تحت افزاری است و scheduling هم باید داینامیک باشه و خروجی به صورت out off order است و مثالش core i3، i5، i7

(نکته: Tomasulo برای اولین بار سال ۱۹۶۷ قبل از این که پردازنده 4004 اینتل وارد بشه، استفاده شده در پردازنده‌ای که IBM ساخته)

مورد ۴ پردازنده‌های VLIW هستند که کاملاً software و static اینجور پردازنده‌ها عمل می‌کنند (static، instruction ها را می‌گیرد و می‌فرستد جلو و هیچ چیز انتخابی نمی‌دونه) هازارد این روش نرم افزاری است و کاملاً باید handle کنه آن را، scheduling آن static است، بنابراین out of order execution هم نداریم (ولی scheduling ما Dynamic باشد)

Scanned by CamScanner

out of execution نداریم

Year:..... Month:..... Day:.....

حالت آخر پردازنده های EPIC هستند (در سال ۱۹۹۰ اینتل و HP آن را طرح کردند و IW یا یک حالت خاصی از EPIC است)، در این حالت issue به صورت static است، هازارد آن نرم افزاری است و scheduling به صورت static است و همه ی hazard detection و این جور چیزها به صورت نرم افزاری handle می شه و اینجوری به کامپایلر این امکان را می دهند که دستورات را دسته بندی کنه و IA-64 کنار هم بذاره و بخت افزار هم دستورات را IA-64 کنه و بفرسته جلو که نوع تجاری از این EPIC پردازنده های Itanium است

اسلاید ۶۹

IW یا چیست؟

مثلا پردازنده فون تا الان دو دونه instruction را می گرفت را اجرا می کرد و می رفت سراغ instruction بعدی instructionها به صورت پیپلاین سر هم قرار می گیرند

IW یا چه می گه پردازنده ی instructionها را مثلا دو تا دو تا، ۴ تا ۴ تا، ۷ تا ۷ تا و ... با هم بخونه و اجرا کنه دستوراتی که مثلا ۵ تا دستور کنار هم در قالب یک دستور طولانی برای پردازنده داده می شه پس ساختار IW یا چه به این صورت است که هر instruction که کامپایلر می خواد تولید کنه از مجموعه دستورات زبان سطح بالای ما یک instruction long است.

long instruction حاوی تعداد مشخصی basic های instruction ما است، آن instruction long هیچ وقت instruction های داخلش از هم جدا نمی شن و همشون در قالب یک پیلیج از حافظه خونده می شن و وارد کش می شن و از کش وارد پردازنده می شن و پردازنده هم این دستورات را همه با هم اجرا می کنه و dependency هم با هم ندارند و خیالش کامپایلر همه ی مسائل را حل کرده است

پس IW یا چه پردازنده ای است که instruction های ش یک super instruction است که در دل خودش چند تا instruction، basic دارد

وقتی می خوان IW یا چه را طراحی کنند یک محدودیت هایی برایش اعمال می کنند، فرض کنیم واحد issue خودش ۵ تا instruction را fetch می کنه، fetch می شن و decode می شن و به عمل FU ها ارسال می شن

نکته مهم اینه که ما FU های متعدد داریم (جمع کننده، ضرب کننده، تقسیم کننده و واحد logic که xor و

Scanned by CamScanner

or و and و ... انجام میشه) که هنگامی که باید به یک قسمت داشته باشیم اینکه وقتی که یک instruction بزرگ داریم

که حد بیش از instruction ، basic وجود دارد باید instruction ها

Year:..... Month:..... Day:.....

مجبوری باشه که چند تا Fu ها را همزمان درگیر کنه یعنی این طور نباشه که یک instruction

very long instruction

بخوریم که توش ۴ تا دستور جمع باشه و هیچ دستور ضرب یا یعنی تقسیمی نباشه ، خب این اینطوری باشه

چی می شه؟ instruction ۴ تا چسبون باید به سمت یک واحد جمع کننده ارسال بشن که خب یک واحد جمع کننده در

هر لحظه از زمان یک instruction می تونه اجرا کنه ولی می تونه ۴ تا جمع همزمان انجام بده ، ما که واحدهای Fu

متعددی داریم ، این مورد را ذکر کنیم ، کامپایلر که آقای کامپایلر تو که می خوای ۵ تا ۵ تا دستور کنار هم برای پیچیدگی

بذاره که دستورات حساب به توش نباشن یا اگر هستند به تعداد Fu های تحت اختیار که من مطلع از اون

دستورات بذاره من دو تا جمع کننده بیشتر ندارم پس در هر long instruction دو تا جمع بیشتر نمی ذاری ،

همچین چیزی را تحت اختیار کامپایلر دیگر می ذنه و کامپایلر زمانی که می خواد long instruction بسازه

از instruction های basic که تولید کرده می خواستش به این صورت در واقع یک مجموعه دو long instruction

توجه داریم و مطابقت داره با تحت اختیار و قابلیت تحت اختیار

مثال: تحت اختیار هر long instruction که بحث من می کنم از این مجموعه instruction ها ۵ تا

instruction اجازه می شه که از اون ۵ تا حداکثر باید یکی branch integer instruction باشه ، دو تا می تونه

Floating point operation باشه و دو تا دیگه می تونه memory refrence باشه ، این مورد را

تحت اختیار کامپایلر دیگه می کنه ، اونوقت کامپایلر باید حواسش باشه هر دستور ۵ تایی که می سازه که

یک long instruction هست بیش از دو تا Floating point توش نداره

* همچین سخت افزار و مخصوصی زمانی می تونه مورد مجوز باشه ولی می تونیم از قابلیتش استفاده کنیم که به اندازه کافی

در کدمان parallelism وجود داشته باشه یعنی ما داریم می گیم که کامپایلر دستورات را مجوزی کار هم

بچینه که بسته های دستوری که داره تولید می شن (بسته های ۵ تایی) باید همواری اجرا بشن یعنی

به ازای هر instruction باید ۴ تای دیگه پیدا کنیم که همواری اجرا بشن و وقتی که کامپایلر داشته هم

رعایت بشه

آیا تضمینی وجود داره که یک برنامه که ۱۰۰۰ instruction داره را ۲۰۰ long instruction خواهد برنامه

کنیم که هر حضش ۵ تا دستور از اون ۱۰۰۰ تا باشه (آیا به ازای هر instruction ، ۴ تای دیگه مثل اون

پیدا می شن که با اون حلو بزن)؟ تضمینی وجود نداره پس کامپایلر هیچ کاری نمی کنه؟ یک دستور جمع داریم

Scanned by CamScanner

که اون دستور جمع وابستگی اش به دستورات قبل و بعدش جوری که نمی‌شه همزمان با هاش ۴ تا دستور دیگه هم جلو فرستاد، در این شرایط کامپایلر مجبوره که اون دستور جمع را تنهایی جلو بفرستد و

Year: ..... Month: ..... Day: .....

long instruction که می‌فرستد جلو و تولید می‌کند که باید ۵ تا دستور basic بسته شوند

یک دستور توش و ۴ تا دیگه دستور nop و پوچ است و این باعث می‌شه که utilization سیستم ما پایین بیاد پس به شرط این VLIW به درد بخور هست که ما هم به اندازه‌ی کافی در زمان موازات داشته باشیم

VLIW به صورت عام محبوبیت نداره ولی در کاربردهای خاص یکی از بهترین گزینه‌هاست

اسلاید ۸۷

یک برنامه که ۲۴ instruction داره و کامپایلر آن را به ۹ long instruction ها تبدیل کرده جاهای خالی این جدول می‌شه کامپایلر سعی کرده دستورات قبل هم پیدا کنه که موازی اجرا بشن ولی پیدا نکرده (دقت ۷ توضیحات هست)

* معایب VLIW

۱) باید به صورت static این parallelism را بین دستورات پیدا کنیم و زحمت را به کامپایلر بدهیم نه به صورت Dynamic و run time انجام بشه

۲) code size ما زیاد می‌شه چون در long instruction ها جای خالی داریم حجم کد ما بالا می‌ره و کل فضای خالی هدر می‌ره

۳) hazard detection به صورت سخت افزاری وجود نداره و زحمات را باید به کامپایلر بدهیم

۴) مهم‌تر از اون اینه که کدها compatibility ندارد یعنی این که اگر مثلاً فرض کنیم اینتل یک پردازنده VLIW ساخته و نشته بوده که من هر long instruction ها ۶ تا دستور توش جا می‌شه اگر خواست ۶ تا instruction نداره و سخت افزارش را زیاد کنه دیگه این کد بدرد نمی‌خوره و باید دور بریزه و متناسب با این long instruction که الان تنظیم می‌شه دوباره کد را بسازد در نتیجه کوچکترین تغییر در آرکیتکچر سخت افزار برای بهبود باعث می‌شه که کدها دیگه قابل اجرا نباشد

Scanned by CamScanner

اسلاید ۸:

* آخرین که الان در پردازنده‌هایی که داریم استفاده می‌کنیم دارن رخ میده این است که می‌شه گفت تمام چیپ‌های تکنیک‌هایی که تا الان

Year: Month: Day:

یعنی را دارن استفاده می‌کنن در پردازنده‌های multi core مثل core i3 که اینتل می‌سازه

پس Dynamic scheduling داریم، multiple issue داریم و speculation هم

داریم

برای multiple issue دو تا راه وجود داره: برای این که بتوانیم در هر کلاک بیش از یک instruction را issue کنیم:

۱) بیایم به جای این که در هر کلاک یک instruction را issue کنیم بیایم در هر سه کلاک یک دستور را issue کنیم و بفرستیم جلو در نتیجه در هر کلاک ۲ تا instruction issue می‌ره

۲) یک ماکزیمم بسازیم که این ماکزیمم کنترلر در هر پریود کلاک کاری نکنه (در هر پریود خروجی چند تا instruction را بتونه هندل کنه (در هر لحظه با multiple istruction روبرو خواهد بود)

در سیستم‌های multiple issue bottleneck issue واحد issue می‌شه. واحد issue است که خروجی instruction ها که می‌بینه تقسیم می‌کنه به سمت چیپ کسی بفرستد، RS ها آیا پر هستند یا خالی هستند، ROB ها آیا جا دارند، وابستگی وجود دارد یا نه. چیپ موری می‌خوایم بکنیم که این RS ها آپدیتش از طرف ROB باید بیاد و این کارها کنترلر واحد issue را پیچیده می‌کنه

اسلاید ۹:

بلاک دیاگرام توضیحات بالا

شبیه شکل جلسه قبل فقط در instruction queue اتفاقا یکی یکی instruction خارج می‌شد

الان instruction خارج می‌شه (یک واحد کنترلی داره که می‌گه هر دستور را در کدوم RS باید بشه

اسلاید ۱۰:

multiple issue:

در حالت multiple issue همه dependency ها در دستوراتی که توی یک دسته قرار می‌گیرند با هم

می‌خوان بررسی بشن باید چک بشن و برای این بررسی می‌شه و این dependency ها باید در RS ها کاف بشه یعنی

در خانه‌های آن RS dependency ها مشخص بشه و برای این که ما CPI کمتر از ۱ داشته باشیم

Scanned by CamScanner

علاوه بر این که باید توسط Fetch، decoder، issue multiple instruction کنیم باید از این طرف هم multiple instruction باید توسط خارج کنیم (شرط این که boundwith بالای 1 یا کمتر از یک CPI باشد شرط لازم این است که باید ورودی بیشتر از یک باشد بعد از آن باید بررسی کنیم خروجی هم بیشتر از یک باشد)

- برای این که allocation RS برای واحد issue راحت تر بشه می شه این محدودیت را روی دستوراتی که توی یک دسته قرار می گیرند اعمال کرد که مثلا هر چهار تا instruction که حساب هم هستند در یک دسته قرار بگیرند یعنی برای این که چند دستور را بین FU ها پخش کنیم باید این پخش شدنی وجود داشته باشد که مثلا بیرون را به جمع کننده بدهیم، بیرون را به ضرب کننده بدهیم، بیرون هم load و store به این که چه تا دستور را چه چهار تا به جمع کننده بدهیم چون کار پیچیده می شه و نمی شه چهار تا دستور را به جمع یک جمع کننده فرستاد

اسلاید ۱۱:

مثال: این مقدار از یک آرایه از یک درایه اش خوانده می شه در r_2 لود می شه و r_2 به اضافه 1 می شه و بعد r_2 را store می کنیم در خط آخر و در عملیات آخر هم شرط loop را چک می کنه

اسلاید ۱۲:

فرض کنیم یک پردازنده multiple issue داریم با این فرض که در هر کلاک حداکثر instruction را می تواند issue کند. مثال اسلاید قبل را با حالتی که speculative نداره بررسی می کنیم (فرق speculative داشتن و نداشتن این که ما سر دستور branch متوقف بشیم یا بریم جلو، حالت No speculative هیچ حالت Tomasulo که سر دستور branch متوقف می شیم)

در این جدول سه بار که اسلاید ۱۱ را اجرا کرده (توضیحات در صفحه ۱۲۰ هست)

اسلاید ۱۳:

چون مثال قبل با speculative است یعنی سر branch جلو prediction جلو می رویم و کار جلو را متوقف می کنیم

یه ستون دیگه اضافه می شه که بعد از write کردن در CDB ما باید در ROB بره تا commit بشه

در حالت No speculative دستورات ما ۱۹ کلاک طول می کشید ولی در حالت speculative ۱۴ کلاک طول می کشه که speed up ما می شه $\frac{19}{14} = 1.3$ که ۳۰٪ درصد بهبود می یابد

Scanned by CamScanner

(Lec7 - chap 3 - G) ریویو هفتم

اسلاید ۴۰

Year:..... Month:..... Day:.....

Branch - Target Buffer

می‌خواهیم بحث اول BTB که اول فصل دیدیم چه‌طوری پیاده‌سازی super scalare است و out of order می‌خواهد که در هر کلاک multiple instruction را به سمت واحد اجرایی ببرد

BTB یک جدول بود که یک فیلد داشت که در آن فیلد آدرس دستورات برنچی بود که تا حالا دیدیم و در اولین اجرای آدرسی است که پیش‌بینی می‌کنیم که آن بیرون pc مازاد یک instruction را از حافظه Fetch می‌کند. هرجا با Fetch کردن میایم این جدول را Look up (جستجو) می‌کنیم و بلافاصله اگر آدرسی در این جدول پیدا کنیم، آدرس مناسب‌تری را به عنوان prediction pc (next instruction) پیشنهاد می‌کنیم و instruction بعدی را از آنجا Fetch می‌کنیم. بنابراین جدول که جستجو سادگی مساوی برقرار بود آن وقت instruction که حدس زده می‌شود به عنوان دستور branch باشد را از حافظه می‌خواهیم در غیر این صورت یا دستوری که داره Fetch می‌شه، دستور branch نیست یا دستوری است که taken نشده که برایش در این جدول آدرس تصمیم‌گیری در نظر نگرفتیم

پس طبق فلوچارت سمت راست زمانی که pc را به سمت حافظه می‌فرستیم instruction را بیاریم با جدول BTB مقایسه می‌کنیم اگر pc در جدول پیدا نشد بعد از این که instruction اومد instruction branch نبود یا Branch بود و Taken نشد هیچ کاری نیاز نیست بکنیم. اما اگر Branch بود که Taken شد آن وقت این دستور Branch را با آدرسش را داخل جدول می‌کنیم و جایی که پریده را set می‌کنیم برای دفعه بعدی.

حالا دوباره برمی‌گردیم به دستوری اول، اگر pc را در جدول پیدا کردیم آن چیزی که جدول می‌ده می‌شه next instruction و pc به خانه‌ای از حافظه اشاره خواهد کرد که ما پیش‌بینی می‌کنیم چون در جدول پیدایش کردیم مانند دستور branch را داریم میاریم، دستور branch رفتی اومد یا Taken می‌شه یا Not taken، اگر Taken شد ما پیش‌بینی کرده بودیم که می‌پره، اگر نپرید یک miss prediction رخ داده و دستوری که به صورت predictive اومده می‌اندازیم بیرون و این آدرسی را هم که پیدا کرده بودیم از جدول پاک می‌کنیم، اگر branch ما Taken شد آن وقت پیش‌بینمون درست بوده و از بلافاصله

Scanned by CamScanner

احتمالی جلوگیری کرده

اسلاید 8

Year: Month: Day:

* یک optimization که روی بحث branch prediction انجام میدن برای این که بازدهی performance بالاتری از پیش بینی بحث Branch Folding است. غیر از این که می‌توان BTB را بزرگتر کرد برای این که پیش بینی را دقیق‌تر کنیم بحث Branch Folding است.

Branch Folding می‌شه که اون جدولی که تا الان آدرس branch را ذخیره می‌کردیم به اضافه‌ی آدرس جایی که می‌خواد بپره، توی این جدول خود instruction Target را هم اضافه می‌کنیم تا الان در جدول آدرس branch بود به اضافه‌ی address Target حالا خود instruction branch را هم می‌برداریم نه آدرس دستور branch که می‌خوایم برا اونجا بپریم (یعنی به جای آدرس، خود اون چیزی که قراره آورده بشه را اینجا بزاریم). چه فایده‌ای داره؟ فایده‌اش اینه که دیگه دستور branch که داریم predict می‌کنیم رو می‌خوا اجرا بشه رو بفرستیم دنبال‌ش دیگه نیاز به Access به حافظه برای آوردن instruction نیست و دیگه نگران این نیستیم که کش miss می‌شه یا نه، نگران اون اتفاقی که Access کش نیستیم، نگران مطلبی Access به کش نیستیم

اسلاید 8

* همه‌ی branch ها شرطی نیستند و بعضی‌ها غیر شرطی اند از جمله branch هایی که Function call می‌کنن و branch هایی که انتهای Function ها return می‌کنن، تمام دستورات return در واقع یک branch غیر شرطی هستند. برای branch های غیر شرطی می‌توانیم به جای این که آن BTB پیچیده با آن prediction ها را داشته باشیم، branch های غیر شرطی خیلی راحت‌تون نرمال‌تره. با توجه به این که اکثر branch های unconditional چون آدرس ها return هستند ما می‌توانیم خیلی راحت Target address را مشخص کنیم (برای غیر شرطی ها) branch های غیر شرطی نیاز به prediction ندارند چون غیر شرطی هستند branch های غیر شرطی همیشه Taken هستند و هیچ وقت not taken نمی‌شن، مسئله سر branch های غیر شرطی اینه که Target address آن کجاست و اگر می‌پره به کجا می‌پره چون آن پریدن متغیر است. یک خط دستوری داریم که ما نشانی را call می‌شد

Scanned by CamScanner

یک دستور branch غیرشرطی به یک جای ثابت همیشگی است (دستوری که یک آدرس را call می‌کند)
دستوری که return می‌کند از یک آدرس، یک دستور غیرشرطی است با مقصد متغیر، اونجاست که
مشکل ساز است: پس دستورات غیرشرطی که فانکشن را call می‌کنند همیشه به یک جا می‌روند و ما حتی نیاز به
prediction هم نداریم و اثر یکبار دستور branch را نمیبینیم و اطلاعات آن را ذخیره نمیکنیم تا آخر برنامه
اطلاعاتش نیاز به تغییر ندارد ولی یک فانکشن وقتی می‌خواد return کند از جاهای مختلفی ممکنه call شده باشه
به جای علف هرز باید برگرده و ما می‌توانیم استفاده کنیم از return stack و دلیلی نداره به همون جای دیگه عمل
برگرده، برای همین روش‌های branch prediction عملی نشاید کلاً این return action ها
بدردنخوره چون هر بار پیش بینی taken درست و چیزی که غلطه آدرس Target address ماست
برای همین بهتر است predictor را عوض کنیم و return address predictor بزاریم و خیلی راحت
با یک ساختار stack می‌توانیم این predictor را بسازیم. می‌دونیم وقتی فانکشن‌ها call می‌شن
به همون ترتیب که call شدن عکس آنها return می‌خورند، پس ما فقط یک ساختار استکی داشته باشیم،
هر instruction که فانکشنی را call کرد return address آن مشخص است و همون آدرس را در
stack ذخیره کنیم بعد آن فانکشن را اجرا می‌کنیم اگر به return برخوردیم از بالای stack برمی‌داریم و اگر
قبل از این که به return برخوردیم خودش یک فانکشن دیگه را call کرد همون لحظه یک return address
دیگه به stack اضافه می‌کنیم

استاتیک:

این نمودار برنامه‌های علف را نشون می‌ده که روی برنامه‌ای انجام شدن که return address predictor
داشته‌اند. با فرض این که سایز این stack که می‌خواد return address را ذخیره کند اگر صفر باشد
درصد miss prediction روی همون خط عمودی است. اگر یک خونه داشته باشیم می‌بینیم که
بعضی جاها miss prediction نسبت به صفر بهتر شده بعضی جاها بدتر شده، اگر دو تا خونه داشته باشیم
miss prediction تقریباً نصف شده، دوباره در ۴ نصف شده و بهتر شده، در ۸ دوباره بهتر شده و در ۱۶ خیلی
با ۸ تا رقابت کرده (این نقطه روی همون ۸ تموم می‌شه)
پس می‌بینیم با یک predictor ساده می‌توان دقت prediction را تا ۹۵ درصد بالا برد

Scanned by CamScanner

اسلاید ۵۸

یک مسئله دیگه در مورد اجرای speculative به وجود می‌آید و آن این که ما چقدر و تا کجا اجرای بی‌راهه

Year: Month: Day:

صورت speculative ادامه بدیم یعنی چی؟ ما یک دستور branch می‌رسیم و این branch

تکلیفش روشن نیست، اجرای speculative می‌شه که یک طرفی را حدس بزنیم و از آن مسیر ادامه بدیم

چقدر طول می‌کشه تکلیف دستور branch مشخص بشه؟ فرض کنیم دستور branch می‌خواد دو تا operand

با هم مقایسه بکنه که دو تا اپرند خروجی دو تا تقسیم کننده هستند و ممکنه ۲۰، ۳۰، ۴۰ کلاک طول بکشه و ما

در این ۲۰، ۳۰، ۴۰ کلاک معطل می‌مونیم و بعد آن چیزی که predictor می‌کنیم جلو می‌رویم و هنوز تکلیف

branch مشخص نشده ما ۴۰ تا دستور بعد از آن را هم اجرا کردیم. ما گفتیم basic block بین

سه تا شش instruction است یعنی ما در هر ۶ تا دستور یک دستور branch می‌رسیم، یعنی

یک branch تکلیفش مشخص نشده سه تا ۶ دستور بعد یک branch دیگه می‌آید چیکار باید بکنیم؟

می‌گیم ما که وقت نداریم خودمون را معطل دستور branch کنیم پس باز هم یک حدس می‌زنیم می‌رویم جلو

دوباره جلوتر یک branch می‌خوریم دوباره حدس می‌زنیم و جلو می‌رویم ....

مشکل چیه؟ اتفاقی می‌افته؟ بعد از ۴۰ کلاک دستور branch اولی، ما می‌فهمیم که پیش بینی ما

اشتباه پیش بینی کردیم آن وقت ۴۰ تا دستور که بعد از این اجرا کردیم همه را باید بریزیم بیرون و

برگردیم از اول یا از این جایی که درست باشه ولی branch های بعدی اشتباه باشه

مثلا در این مسیر بر اساس حدس یک cache miss رخ داده باشه، یه ارور هم چیزی را پیاده و ... که

همش بر اساس حدس است و اگر اشتباه پیش بینی کرده باشیم انگار یک کاری را انجام دادیم که اینجا بحث

که ممکن است prediction منجر به performance degradation شود و منجر به این شود که

کلاک بیشتری از پردازنده خارج شود و performance ما افت کند در کنار این که power و انرژی را هم

از دست دادیم اینجا بحث که باید بررسیم تا کجا عمق این prediction را باید افزایش دهیم؟

پس speculation مجبور می‌شود energy efficient باشه که performance خیلی خوبی

به ما بده یعنی با prediction جلوی بخش زیادی از ریسک‌ها را گرفتیم در این حالت هم performance

خوبی داشتیم هم انرژی خوبی داشتیم

Scanned by CamScanner

اسلاید ۹:

این نمودار اجرای برنامه های مختلف و نسبت miss speculation است (مشخصه چه چیزی)

Year:..... Month:..... Day:.....

اسلاید ۱۰:

* یک prediction دیگه داریم به نام value prediction. فرض کنیم یک instruction داریم که مقدار یک operand است که مثلا دستور تولیدش نکنه (وابستگی RAW) مثلا فرض کن یک دستور جمع مقدار از دستور تقسیم است، value prediction می‌تونه جمع مقدار تقسیم شده برو جلو و یک دستور برای خودت فرض کن و با اون کار خودت را انجام بده. فرقش اینه که خروجی تقسیم چیز دیگه ای است و می‌ری از اول انجام میدی و اگر همین بود که فرض کردی خیلی سود کردی

می‌تونه در مسئله یک دستور خروجیش همیشه یکسان نیست ولی می‌تونه خروجیش را در یک رنج مشخصی برحسب دستورات گذشته پیش بینی کرد. مثلا یک دستور branch می‌خواد چک کنه عدد a مثبت است یا منفی. اگر این a را توسط یک دستور تقسیم همیشه تولید می‌کردیم، این دستور تقسیم ده بار اخیر این a را تولید کرده و خروجیش مثبت بوده، آیا انصاف نیست که این دفعه هم به branch بگیم تو a را مثبت بگیر و برو جلو

predict کردن به صورت تخمینی سخت نیست ولی به صورت دقیق سخت است پس اگر ما بخوایم محاسبات دقیق انجام بدیم و محاسبات ترتیبی بخوایم انجام بدیم شاید با value prediction بتوانیم سرعت را زیاد کنیم ولی اگر بخواد به صورت دقیق رفتار کنه کار سختی است

تحت value prediction جایی که می‌تونه کاربرد خوبی داشته باشه در بحث آدرس‌هاست یعنی address prediction انجام بدیم. گاهی استفاده می‌شه. ما در دستورات load و store آدرس داریم که از فلان جای حافظه که مقدار آدرس آن حافظه را این رجیستر مشخص می‌کند دیتا بخون بیار در رجیستر a. محتوای این رجیستر را به فلان خونه حافظه که آدرسش را در فلان رجیستر نگه داشتیم یا آدرسش را توسط فلان عملیات جمع و تفریق تولید می‌کنیم و در فلان رجیستر می‌ذاریم اونجا ذخیره کن

در بحث load و store خیلی وقت‌ها dependency بین آنها پیش می‌آید یعنی فرض کنیم دو تا دستور store داریم که دو تا دستور store مستقل از هم دو تا آدرس برای target خودشون دارن که این آدرس‌ها value مشخص نیست. آیا این دو تا store را می‌تونیم جابجا کنیم؟

Scanned by CamScanner

store دوم می‌تواند جابه‌جا شود اگر آدرس مقصد این دو تا دستور یکسان نباشد

Year: Month: Day:

یا یک دستوری را می‌خواهیم load کنیم (یک رجیستر را از یک جای حافظه load کنیم)، یک دستور store هم داریم که می‌خواهد یک رجیستری در یک خونه حافظه بنویسد، دستور load بعد از store اگر هر دو دستور load و store، value آدرس مشخص نیست چون در حال رجیستر است اما load از جایی می‌خواد بخونه که store می‌خواد در آنجا بنویسد پس load نمی‌تونه زودتر از store انجام بشه، حالا اگر بنویسیم پس نمی‌کنیم که value آن آدرسی که در load و store هست رو بنویسیم این اجتناب خطر را با دقت بالایی داشته باشیم که این دو تا آدرسشون با هم یکی هست یا نه prediction انجام بدیم می‌توانیم out of order execution را بالا ببریم را از لحاظ دستورات جابه‌جایی کنیم

«ویدیو اول» (Lec6-1)

اسلایدها:

- data level parallelism در رابطه با SIMD صحبت می‌کند این پردازنده‌هایی می‌توانند انبوهی از دیتاها را با یک instruction parallel پردازش کنند این پردازنده‌ها مناسب اپلیکیشن‌هایی هستند که پردازش موازی دارند مثلاً عملیات جمع روی دو تا ماتریس، n تا دیتا کار یکسان جمع باید انجام شه که این ها به صورت موازی می‌تونه پردازش بشه، یا اپلیکیشن‌های پردازش صوت و تصویر که انبوه دیتاهایی دارند که می‌تونه به صورت همزمان و موازی انجام بشن پس چرا این اپلیکیشن‌ها را به صورت موازی انجام ندهیم تا performance بیشتری بگیریم

* SIMD در مقابل MIMD برای پردازش می‌تواند energy efficient تر باشد از MIMD. این مقایسه از این جهت هم است که نسبت مثال‌ها بیشتر بیاد که SIMD نوع خاصی از MIMD نیست و وقتی MIMD داریم دیگه احتیاج به SIMD داریم؟ MIMD که تابع‌هاش بیشتر است پس چرا ما SIMD نیاز داریم؟ نکته‌ای که وجود داره SIMD وقتی می‌خواد multiple data را پردازش کند همه‌ی کار instruction را اجبار می‌کند به این که مثل MIMD، n بار instruction را انجام دهد.

از طرفی از این سرعت تا اثر اون multiple که داریم در SIMD با اون m که داریم در MIMD است پس باشد یعنی میزان موازات برابر باشد performance یکسانی خواهیم گرفت اما از طرفی صرف انرژی کارت زیاد می‌شه به این معنی که وقتی یک instruction را یکبار بخوانیم، یکبار از حافظه خوندیم و یکبار Fetch و decode و exe کردیم هر چند که execution unit

Scanned by CamScanner

Functional unit - m بار باید تیا را پردازش کند (از نظر process روی دیتا فرقی ندارد ولی هزینه‌ای که برای execution آن دستور صرف می‌شود در SIMD به مراتب کمتر است)

Year: Month: Day:

پس SIMD در ساختارهایی که موازات دیتا وجود دارد مناسب است که یکی از آنها mobile device هستند. موبایل‌های جای پردازش تصویر و صوت زیاد دارند

- مزیت دیگر SIMD به MIMD این که برنامه‌نویس نیاز نیست که تفکر یا ذهنیت را برنامه موازی کند و می‌تواند برنامه را سریال بنویسد و به صورت sequential بنویسد و نیاز نیست multi threading بطلب SIMD انجام بدهد و ساختار برنامه را هم بریزیم، در ساختار SIMD برنامه‌نویس همچنان به همون نظر سنتی خودش ادامه می‌ده اما کامپایلر می‌دونه این برنامه sequential را برای چه سختی باید کامپایل می‌کنه (برای چه پردازنده‌ای چه ساختاری) بنابراین آن جاهایی که امکان موازات در آن وجود دارد single instruction می‌تواند mutiple data را به صورت موازی پردازش می‌کند آنجا instruction هایی تولید می‌کند که مناسب همین کار باشد. یک پردازنده SIMD در دل خودش SISD را دارد

اسلاید 3:

*- پردازنده‌های SIMD سه نوع از آنها وجود دارد:

1) vector architecture (vector process)

2) SIMD extension در پردازنده‌های SISD رایج که استفاده می‌کنیم

3) Graphics processor unit (GPUs)

- در پردازنده‌های x86

* به ازای هر سال در chip پردازنده دو تا core اضافه شد

* اون قسمت SIMD extension از راهی که طراحی شده هر ۴ سال یکبار عرض و دیتاهایی که می‌توانستند از هم پردازش شوند دو برابر می‌شده

* انتظار می‌رود speedup که از قابلیت SIMD پردازنده‌های x86 به وجود می‌آید حتی تا دو برابر قابلیت multi core شدن آن باشد مثلاً پردازنده‌های اینتل در x86 می‌توان این انتظار را داشت به شرطی که عرض SIMD که سایز آن می‌شد توسط x86 دو برابر می‌شد performance که خاطر این قابلیت اضافه شده می‌بینیم بیش از performance باشد که core ها را دو برابر کنیم

Scanned by CamScanner

اسلاید ۴:

Year:..... Month:..... Day:.....

* vector processor:

- یک الگوریتم به نام Jim smith می‌گوید که efficient ترین راه برای اجرای الگوریتم‌های vectorizable یا قابل برداری بودن باشد یعنی vector processor ها بهترین گزینه است برای الگوریتم‌هایی که قابلیت vector دارند (vector می‌شوند آرایه و ماتریس و ... جایی که با انبوه داده‌ها صورت موازی می‌توان پردازش شوند و داده‌های برداری داریم)

- ایده اولیه پردازنده‌های برداری:

در پردازنده به جای این که رجیستر‌ها را دونه دونه مدیریت کنیم vector register داشته باشیم یعنی n تا رجیستر که با هم داده‌ها توشون ریخته می‌شه و از اون ور هم پردازش می‌شوند. n تا رجیستر می‌تونن با یک instruction پردازش شوند و قسمت جالب هم می‌تونیم result هایی که تولید می‌شه رو حافظه بریزیم

- این رجیستر‌ها سخت‌افزار زیادی مدیریت می‌دارد. وقتی پردازنده vector processor هست و به صورت vector پردازش کند پس کامپایلر باید این instruction ها را برایش تولید کند پس کامپایلر هم باید این vector register های که قراره داده توش بیاد و قراره یک instruction هم Fetch و decode و روی این register ها یک کاری را انجام بدهد باید تعیین کند و وقتی ما با استفاده از کامپایلر دستورات برای تولید می‌کنیم می‌توانیم امیدوار باشیم که از فضای باز حافظه هم استفاده را کرده‌ایم به این دلیل که هایک دستور را Fetch می‌کنیم و این دستور ما حیله که برد ۳۲ بایت یا ۶۴ بایت باید داده را از این جای حافظه به این هم هم داده بیاره و هاست هم ~~[illegible]~~ داده‌ها را دریافت می‌کنیم روی bus های سیستم و این حجم bus ها باید ارای یک تک داده‌هایی که خوانده شده مجبور نیست دستور load اجرا شه، بلکه دستور به حافظه بره، بلکه حافظه یک داده را بیاره و دوباره به حافظه دستور بدهیم و داده‌ها را از آن بگیریم و این تعاملات بین پردازنده و حافظه بسیار را انجام می‌شه و n تا داده حرفه‌ال می‌آید و این باعث می‌شه که از فضای باند که داریم بیشتر استفاده و overhead های جدا کمتر بشه

اسلاید ۵:

* مثالی یک نوع از vector processor:

فرض کنیم یک پردازنده داریم به نام RV64V (risc 64 vector extension)

این پردازنده vector register های ۶۴ بیتی داره و این vector register تعدادشون ۳۲ تا است و این مجموعه رجیستر‌ها که بهشون register file می‌گویند حداکثر ۱۶ تا برای read داره به اضافه n تا برای write

Scanned by CamScanner

Fu ها Fu برداری است و ظاهراً پایپلاین شده است و به صورت پایپلاینی می‌توند انجام بشه و این Fu ها و control hazard ها می‌توانند detected شدن و واحد load و store هم برداری است و پایپلاین شده و در هر کلاک یک word را می‌توند تولید کند

Year: Month: Day:

در کنار این واحدهای vector رجیسترهای scalar هم دارد (رجیسترهای عادی) :

۳۱ تا رجیستر عام منظوره داره + Floting point ۳۲ تا رجیستر

اسلاید ۶:

مثلاً رکاب توضیحات مثال :

یک Floting point Fu جمع کننده ترین کننده داره، یک ضرب کننده، یک تقسیم کننده داره یک integer unit و logical unit داره و یک قسمت رجیسترهای scalar داریم که همون یک رجیسترهای هستند که درون دیتا ها می‌توند توشون ذخیره بشه به اضافه vector register که قسمتی ۳۲ تا است و هر کدوم رجیسترهای ۶۴ بیتی می‌توند داشته باشند و واحد load و store داره که به صورت برداری عمل می‌کنه یعنی یک دستور load برداری را به این قسمت vector load/store می‌دیم و این دستور load یا store برداری خودش می‌آید دستور می‌ده به حافظه که این n تا دیتا را بیت سر هم برایش مدیریت می‌کنه که بعضی در vector register ها

اسلاید ۷:

مثال: برداری نده RV64V چند نوع عملیات جمع و ضرب و ... در Fu می‌توند انجام بده که هر کدوم چند تا ورودی داره مثلاً عملیات جمع را در نظر بگیریم می‌توند هم جمع integer داشته باشیم هم جمع floting point اون جمع FP می‌توند دو تا دیتای scalar با هم جمع بشه، می‌توند یک دیتای scalar با یک vector جمع بشه یا هر دو operand به صورت vector باشند پس ورودی های مختلفی از دستور add داریم و وقتی به برداری نده دستور add می‌دیم و Fetch و decode می‌شه موقع decode شدن این دستورات add با هم متفاوت هستند برای control unit ها فضای متفاوت دارند و عملیات متفاوتی باید انجام بشه.

اگر هر دو operand ها vector باشند VV را دستورهایی جیسون مثلاً VV add

اگر اولی scalar و دومی vector باشه VS به دستورهایی جیسون یا اگر برعکس باشه VS به دستورهایی جیسون

Scanned by CamScanner

دستور load و store هم که هم scalar داریم هم vectore که به صورت LS یا VS هستند

Year:..... Month:..... Day:.....

حالا می‌خواهیم مثال $Y = a * X + Y$ را انجام بدیم که X و Y که به صورت بردار نوشته شده‌اند برداری هستند و a به صورت scalar است که به اینجا DAXPY گویند

حالا می‌خواهیم بگیم این دستور را در یک پردازنده scalar و مرحله بعد پردازنده vector processor SIMD بنویسیم

گفتیم X و Y برداری هستند پس باید $Y + X$ در loop ها نوشته بشه.

اشاره می‌کنه به ابتدای بردار X

آرایه‌ها ۳۲ تا درایه دارند که این المان‌ها ۸ بایت هستند

حالا a را لود می‌کنیم در رجیستر F_0 و بعد x_{05} جمع می‌شه با ۲۵۶ می‌ایستیم در x_{28}. بعد می‌اومدیم در loop. بعد از جایی که x_{05} می‌شه می‌ایستیم در F_1. بعد F_0 را ضرب کن در F_1 (در F_0 مقدار a هست) و بعد بریزیم در F_1. بعد لود کن Y را که از x_{06} در F_2 بعد F_1 را با F_2 جمع کن بریز در F_2 بعد F_2 را store کن به

اشاره می‌کنه به ابتدای بردار Y

همون جایی که Y_1 را از آنجا برداشتیم. بعد هم x_{05} و x_{06} را جمع می‌کنیم با ۸ تا برد به درایه بعدی (x_{05} و x_{06} علاوه دو تا رجیستر یکسان هستند) در نهایت x_{05} با x_{28} کاهش می‌شه که نسبیم یا به آخر loop ها رسیدیم یا اگر نرسیدیم دوباره به اول loop می‌ریم و ۳۲ بار کار را اجرا می‌کنیم

حالا اگر بخواد روی vector processor که رجیستر بود که R_{64} بود بخوایم همین کار را بکنیم vector processor ها ۳۲ بیتی هستند یعنی عملیات vector که داریم روی ۳۲ تا داده کار می‌کنند.

دستور اول می‌گه ۴ تا vector register ها که FP هستند و ۶۴ بیتی هستند را enable کن و بریز در مود vectore. برای دستوری که به صورت vector می‌خوایم انجام بدیم باید این کار را بکنیم که به تعداد رجیسترها باید vectore register را enable کنیم و بعد کار اصلی را انجام بدیم

کار دوم اولش اینه که a را بریزیم در F_0. بعد x_0 را لود کنیم در V_0 ولی load ما به صورت V_0 است یعنی vector load انجام بدیم از جایی که x_0 داره شروع می‌شه به vector register V_0. پس یک vector x به اندازه‌ی ۳۲ تا ۶۴ بیتی را load می‌کنیم در V_0 (بدیک کلاک چند تا این کارها نمیشه انجام مثلا در ۳۲ کلاک انجام می‌شه). حالا کافیه یک ضرب برداری انجام بدیم و V_0 را ضرب کنیم در F_0 بعد y را می‌خوایم از آدرس V_1 در رجیستر V_2 می‌آوردیم پس ۳۲ تا ۶۴ بیتی خوانده می‌شه و در V_2 می‌آید (خب V_2 حاوی کل y است، V_1 هم حاوی $a \times X$ است) حالا باید V_1 را با V_2 جمع کنیم در V_3 بریزیم و به صورت vectore باید عمل بشه و در نهایت هم V_3 را باید ذخیره کنیم در جایی که V_1 می‌ره و ما دیگه اینجا loop نداریم چون همه این دستورات که V_1 اولشون آمده به صورت loop که

Scanned by CamScanner

۳۲ تا Iteretion داره عمل می‌کنه. کارمون هم که تموم شد مد vectore بودن را disable می‌کنیم
و دستورات بعدی دستورات برداری نیستند

Year:..... Month:..... Day:.....

در حالت اول که به صورت scalare نوشتیم ۸ تا دستور را باید در یک loop ۳۲ بار اجرا می‌کردیم به اضافه
دو تا instruction که اولش بود پس ۲۵۶ بار این instruction را اجرا می‌کردیم برای این که پردازشی را
روی دستور ۳۲ بیتی انجام بدیم

در حالی که اگر به صورت برداری عمل کنیم ما به جای ۲۵۸ تا دستور، ۶ تا دستور اجرا کردیم به اضافه یک دستور
اول و یک دستور آخر که مجموعاً ۸ تا دستور داریم (هر چند که نمی‌شه بعضی از دستورات مثل vmul را هم توهم
خیلی معادل fmul بدونیم چون vmul داره ۳۲ تا عمل ضرب انجام می‌ده ولی fmul یک عمل ضرب انجام می‌ده
که قطعاً vmul زمان اجرای طولانی‌تری داره ولی یک بار فقط اجرا می‌شه ولی ۳۲ بار ضرب کننده ضرب را
انجام می‌دهد ولی در کل خیلی کمتر از حالت scalar هست)

اسلاید ۸

* عملیات vectore زمان اجراشون به سه تا پارامتر وابسته است:

۱) اون operand که برداری است طولش چقدر هست. operand های ما نمی‌تونند ۳۲ تا نیستند یا
دقیقاً ضریبی از ۳۲ نیستند ممکنه ما ۱۶ تا داده داشته باشیم در این صورت نمی‌تونیم از تمام قابلیت های vectore
استفاده کنیم یا مثلاً ضریبی از ۳۲ نباشه مثلاً ۴۰ باشه که اول باید ۳۲ را اجرا کنیم بعد ۸ تا رو اجرا کنیم
و زمان اجرای ما قطعاً بیشتر خواهد بود که طول vectore های ما چند برابر باشه

۲) structural hazard است که قسمتی وجود دارد یعنی ما نمی‌تونیم آنرا روی واحدهای محاسباتی کم می‌آوریم
جمع برداری که ما می‌تونیم ۳۲ تا جمع را همزمان انجام بدیم اگر ما فقط یک جمع کننده بیشتر نداشته باشیم
پس مجبوریم ۳۲ تا جمع را sequential داشته باشیم

۳) مشکل data dependency هم وجود می‌آید که می‌تواند مانعی باشد که مجبور باشیم
دستورات را متوقف کنیم یا اندکی تأخیر بیاندازیم

- فرض ما اینه که Fu های ما می‌تونند در هر کلاک یک المان یا دیتا را بفرستند جلو یعنی هم
نسبت ضرب کننده یا تقسیم کننده چند کلاک طول می‌کشه چون پایپلاین هست این قابلیت را داره که در
هر کلاک می‌تونه یک دیتا را بخواند

Scanned by CamScanner

- convoy ← به مجموعه از vector instruction ها می‌گه به صورت بالقوه می‌تواند با هم جلو اومدن یک convoy کنیم. یک دستور برداری خودش به تنهایی n تا دیتا را می‌خواهد پردازش کند.

Year: Month: Day:

convoy

می‌بینید در دنیا این یک دستور برداری ما چقدر می‌توانیم بکشری دستورات دیگه بسازیم. هم ترازهم که از ما هم به موازات این اجرا شن یعنی جمع کننده دارد یک vadd انجام میده و ۳۲ تا جمع را پشت سر هم داریم می‌فرستیم به سمت یک واحد جمع کننده و ضرب کننده که اون در یک کار هست را اجرا است تا دو تا هم بیاید بین دستور بعدی که mul است یک یا دو تا vector register دیگه را بگیریم از operand های عملیات ضرب به موازات عملیات قبلی ضرب کننده هم عملیات ضرب را انجام دهد. اگر data dependency و structural hazard وجود نداشته باشد می‌توانیم این جمع را موازی کنیم با اون ضرب به اصطلاح می‌گیم اینجا در یک convoy قرار گرفتند این convoy را گفتیم که در آینده بتوانیم زمان اجرا را تخمین بزنیم

اسلاید ۹

- یک اصطلاح دیگه chime است.

- قبل از اون chaining را می‌گیم (یعنی زنجیره‌وار اجرا کردن دستورات): فرض کنیم که یک دستور add داریم که ۳۲ تا دیتا را می‌خواد با هم جمع کند. یک دستور ضرب داریم که ۳۲ تا دیتا را می‌خواهد در هم ضرب کند. یا چون مثال $a * X + y$ که ما باید a را ضرب کنیم در ۳۲ تا المان x_i ها بلد که تموم شد یک اینجا را جمع کنیم با y_i ها. اگر یک جمع کننده و یک ضرب کننده بیشتر نداشته باشیم $a \times X_i$ ها حداقل ۳۲ تا کلاک طول می‌کشد اما اینجا یک وابستگی RAW داریم. Read دستور جمع ما در جمع کردن y با $a \times X$ باید زمانی انجام بشه که write دستور $a \times X$ انجام شده باشه. وقتی که $a \times X_0$ شد و اولین المان تولید شد می‌توانیم عملیات جمعی که حداقل RAW دارند را شروع کنیم برای y_0 یعنی y به محض این که $a \times X_0$ شد می‌توانیم جمع بشه با حاصل ضرب. این وجود نیست منتظر بمونیم a را در یک یک المان های X تولید کنیم دستور برداری $a \times X$ و دستور جمع برداری y با هم وابستگی RAW دارند که با یک chaining اینجا را پایپلاین می‌کنیم و add ما فقط یک کلاک از mul ما عقب می‌افتد به خاطر این که مجبور باشه ۳۲ کلاک انجام بشه تا کامل عملیات $a \times X_i$ ها انجام بشه

Scanned by CamScanner

chime: به واحد زمانی که برای اجرای یک convoy نیاز است (convoy مجموعه از دستورالعمل‌هایی است که می‌توانند موازی اجرا شوند) مثلاً این دستور axX و بعد y که یک بلاک از هم عقب بودند می‌توانند در یک convoy قرار بگیرند و با هم اجرا شوند فقط باید اختلاف. حالا زمانی که طول می‌کشد تا این عملیات convoy تموم بشه یعنی همه‌ی جمع‌ها و ضرب‌ها در این vector تموم بشه می‌گیم یک chime. این یک چیزی به راحتی می‌ده از زمان اجرای برنامه‌ها، اگر بدونیم یک برنامه چند convoy داره و می‌دونیم هر convoy یک chime طول می‌کشه می‌تونیم اونوقت می‌تونیم بگیم چند کلاک طول می‌کشه. پس m تا convoy اگر داشته باشیم m تا chime هم داریم برای vector عملیاتی که طولش n است. و هر chime هم می‌تونیم بگیم به اندازه‌ی n کلاک طول می‌کشه یعنی یک ضرب. و کتوری توی ۳۲ تا المان ۳۲ کلاک طول می‌کشه. پس اگر برای vector به طول n، m×n کلاک طول می‌کشد

اسلاید ۱۰:

یک مثال:

لود کنیم از x_5 به وکتور رجیستر v_0، v_0 را ضرب کنیم در یک رجیستر f، باید لود کنیم از x_6 به v_2، بعد v_2 را با v_1 را جمع کنیم بریزیم در v_3 و بعد v_3 را store کنیم در جایی از حافظه که x_6 اشاره می‌کند. می‌خوایم ببینیم چند تا convoy می‌شه و چقدر طول می‌کشه

این ۵ تا دستور رسته تا convoy می‌شن. یک دستور لود داریم که در کلاک اول می‌آید. ضرب بلدش هازارد RAW داره با دستور لود بالاش اما با استفاده از تکنیک chaining ما مجبور نیستیم mul را انقدر معطل کنیم تا لود تمام تموم شه بلکه می‌تونیم به صورت chaining با هم اجرا کنیم ولی mul یک کلاک عقب‌تر از لود باشد

دستور بعدی یک لود است این لود در بلوک مثلاً با لود قبلی هیچ وابستگی ندارند و هازارد وجود ندارد اما می‌تونیم همزمان با هم اجرا کنیم اما structural hazard دارند با هم این دو تا لود، اما هفت افزار ما بیشتر از این کشش نداره و وقتی یکی داره لود می‌کنه کس دیگه‌ای نمی‌تونه لود کنه (در هر زمان فقط یک لود می‌کنه و اگر اول loader قابلیت پیشبری هم داشته باشه مثلاً ۵ تا عملیات لود اول را سریع‌تر اجرا کنه چه کاری میوه می‌زاغ لود دیگه‌ای) پس اول لود بعدی می‌ره در convoy دوم، دستور add به mul وابستگی دارند و هازارد RAW دارند اما اینجا می‌تونیم با استفاده از تکنیک chaining این add را با دستورات convoy اول بفرستیم چون اینا v_2 را داره که توسط لود دوم تولید می‌شه و چون اول لود را در convoy دوم قرار گذاشتیم می‌شه این را در convoy اول بذاریم و این add باید بعد از ~~convoy~~ لود دوم باشه پس در convoy دوم قرار می‌گیره

Scanned by CamScanner

دستور آخر store است که می‌خواد store کند ۲ را که توسط add تولید می‌شه. اما این store با add یک convoy
قرار گیره و خب store هم قبل بوده است و وقتی bus convoy قبل کاملا مشغول بوده است

Year: Month: Day:

store می‌تونه در آن convoy قرار گیره (می‌دونیم که store و load توسط یک واحد ALU اجرا می‌شن)
خاطر همین store در convoy سوم طول می‌کشه. در نهایت عملیات ما در ۳ chime تموم می‌شه و
در نهایت دو تا floating point operation انجام دادیم یعنی مجموعه‌ای از هر یک عملیات result
که تولید می‌شه دو تا Fpo داریم (چون هر یک result یعنی یک store که انجام بدیم و هر یک store که
انجام بدیم حاصل ضرب یک ضرب Fp و یک جمع Fp است) در نهایت Flop را بخواهیم حساب کنیم یعنی
بدونیم به ازای هر floating point operation چند کلاک صرف کردیم، اگر سه را گفته بخواهیم حساب کنیم
به ازای هر Fpo می‌شه ۱۵ کلاک چون مجموعا ۳ chime طول کشه و این chime ها به اندازه‌ی طول
vector طول کشه به ازای هر یک دیتا که تولید شده دو تا Fpo انجام شده در نهایت اگر فرض کنیم هر یک دیتا در
یک کلاک طول می‌کشه به ازای هر Fpo ها ۱۵ کلاک صرف کردیم
در نهایت کل این عملیات چقدر طول می‌کشه؟ اگر فرض کنیم که vector ها ۶۴ تا آیتم دارد و هر chime،
۳۲ کلاک طول کشیده $۳۲ \times ۳ = ۹۶$ کلاک طول کشیده
۳ تا chime داریم که هر chime، ۳۲ کلاک طول کشیده

اسلاید ۱۱

تا اینجا فرض کردیم رجیستر ها وکتوری هستند اما FU ها وکتوری نیستند. اگر بخواهیم به صورت وکتوری عمل کنیم
طبق ساختار سیستم جلو پیش می‌رفتیم که یک FU اگر می‌خواد روی n تا رجیستر جمع را انجام می‌دهد به این صورت
vector رجیستر ها را می‌چیدیم و A[1] + B[1] ، A[2] + B[2] ، ... جمع می‌شن و در هر کلاک یک خروجی داریم
اما دو مشکل داریم performance بیشتری بدیم و اگر به ازای n تا دیتا یک instruction را اجرا می‌کنیم اول
n تا دیتا هم باید برای هر دیتا n بار اجرا بشن پس به جای یک واحد جمع کننده مثلا ۴ تا جمع کننده می‌ذاریم که این
۴ تا به صورت موازی عمل می‌کنند و به صورت سطری رجیستر ها را پر می‌کنیم
اگر بخواهیم به صورت فنی‌تر بررسی کنیم می‌شه شکل سمت راست که فرض کنیم vector registers های داریم که
در چند line کنار هم قرار گیرند و یک FU ضرب کننده و یک FU جمع کننده در هر باند داریم و با کمک از طریق تکنیک chaining
استفاده می‌کنیم و همزمان با هم ۴ تا جمع را در این وکتور رجیستر های هر لاین می‌ریزیم و وقتی حاصل جمع تولید شد
عمل ضرب می‌تونه انجام بشه. این ساختار سخت‌افزار ما را پیچیده‌تر می‌کنه ولی سرعت ما را بیشتر می‌کنه
و از این ساختارهای chime و chaining و ... این ساختارهای چند باندی می‌تونه این زمانی که به ازای هر chime

Scanned by CamScanner

در نظر بگیریم را که هشت دهم بین یک convoy در زمان کمتری می‌توان با این ساختارهای موازی اجرا است

Year:..... Month:..... Day:.....

اسلاید ۳۱۲

عملیاتی که روی دیتاها می‌کنیم لزوماً روی دیتاهای مجاور نیست و ممکن است عملیاتی که روی یک وکتور انجام می‌شود نیست یعنی اگر یک load ۳۲ تا رجیستر می‌خواد جمع کند ممکن است تعداد عملیاتی که روی یک وکتور انجام می‌شود بیشتر از این ۳۲ تا باشد در این صورت چه جوری می‌توانیم برنامه‌ای بنویسیم که مجموع درست انجام بشه یعنی وقتی می‌گیریم load کردن انجام دادیم به سخت افزار می‌گیم ۳۲ تا رجیستر را با هم جمع کن در ۳۲ تا رجیستر مقصد بریز. وقتی بحث می‌کنیم store کردن یعنی می‌گیم ۳۲ تا رجیستر را مثلاً جای خاصی store کن ممکنه store ما ۳۲ تا رجیستر نداشته باشه و وقتی وکتورهامون ۴۰ تایی است باید اول ۳۲ را لود کنیم و جمع کنیم و store کنیم و در مرحله بعد ۸ تایی را لود کنیم و جمع کنیم و store کنیم یا اگر ۳۲ تا باشیم معمولاً از ۸ تایی استفاده می‌کنیم و بقیش دیگه خالی می‌موند و مشکل اساسی اینجاست که خیلی وقت‌ها طول vector store ما در زمان کامپایل به صورت static و کامل این مشخص نیست و طول وکتورها مشکل است به صورت آنلاین یا در زمان اجرا مشخص شود و می‌توانیم با برنامه‌نویسی قبل از کامپایل مشکل طول وکتورها را حل کنیم برای همین vector processor ها یک vector length register دارن که باید به این مقدار دستی بدهیم و با کمک آنها control unit برداننده فهمیدیم که این دستور وکتوری که الان داریم به شما می‌دهم می‌خواد از تمام قابلیت‌های وکتوری شما استفاده کنه و اگر یک instruction وکتوری شما ۳۲ تا دیتا را process می‌کنه این instruction من ۲۰ تا دیتا داره و خواست به این مسئله باشه و فقط ۱۵ تا دیتا را process کن و ۱۲ تای دیگه چیزی توش نیست

اسلایدهای

خب بیایید در اینجا همین توضیحات بالا را می‌زنیم که می‌خوایم عملیات $a * x[i] + y[i]$ کنیم. در حال این n دیتا ۳۲ است و معمولاً ما یک بار اجرای یک دستور ضرب وکتوری این ضرب انجام می‌شه یا یکبار جمع وکتوری این عملیات جمع انجام می‌شه و یکبار store وکتوری ذخیره می‌شه ولی به صورت general این n هر عددی می‌تونه باشه که ما با این روش می‌خوایم کدش را بنویسیم

پس باید یک loop بنویسیم که در آن loop در هر iteration ۳۲ تا از این دیتاها با پردازش بشه تا iteration آخر شه که دیگه ۳۲ تا نیست و باقیمانده ۱۱ است پس ۳۲ مثلاً اگر ۷۵ تا باشه در دور دوم ۳۲ تا انجام می‌شه پس ۶۴ تا در دور آخر ۶ تا اجرا می‌شه و در loop باید این مسئله را handel کنیم که یکی در دور آخر هستیم و ۳۲ تا دیتا با باقیمانده آنها پردازش بشه که با استفاده از vector length processor این مشکل حل می‌شه توضیحات کلاس ۱:۰۵

Scanned by CamScanner

اسلاید ۸۱۳

8 vector mask register

Year:..... Month:..... Day:.....

در یک loop ما می‌خواهیم $X[i] = X[i] - Y[i]$ را محاسبه کنیم و ۶۴ بار می‌خواهیم این کار را انجام بدهیم به جهت بالا بردن بازدهی عملیات وکتوری که ۳۲ تا داده را می‌توانند وکتوریکه این کار را می‌توانیم انجام بدهیم اما یک ایراد این روش خاصیت که ما گفته به شرطی انجام بده که اگر $X[i]$ صفر نباشد پس عملیات روی همه‌ی درایه‌ها انجام نمی‌شود و حتماً ۶۴ تا عملیات انجام می‌شود پس می‌توانیم با یک عملیات sub وکتوری انجام بدهیم چون مثلاً ۳۲ تا X را همه‌ی ۳۲ تا را می‌کند در ۳۲ تا X می‌ریزد و کاری ندارد از این ۳۲ تا یکدوتا صفر است یکدوتا صفر نیست پس باید به جوری جلوش بگیریم که عملیات را روی همه انجام نده که با استفاده از vector mask register جلوش می‌گیریم.* vector mask register

می‌گه که n تا ایلان را که لود کردی مثلاً هم یک رجیستر n بیتی مثلاً ۳۲ بیتی جلوت می‌دهم که این رجیستر ۳۲ بیتی هر بیتش نشان‌گر این است که روی این درایه این کار انجام بشود یا نه. و جایگاه این Flag یا یک بیت صفر بود کاری نداشته باش و اگر یک بود کار را انجام بده

مثال: ۳ تا از وکتور رجیستر‌ها را درست می‌کنیم و بعد با دستور خط دوم vectore mask register را Enable می‌کنیم تا عضو به عملیات را mask کند. لود می‌کنیم v_1 که شرط $X[i]$ است و می‌گه می‌ریزیم در v_2 و v_3 که y و x است و می‌گه می‌ریزیم در v_4. بعد شرط v_0 را more می‌کنیم به f_0. v_0 یک رجیستری است که ما از قبل فرض می‌کردیم که به معنی این شرط f است. بعد می‌گه اگر f_0 v_0 not equal باشد وکتوری not equal در f_0 ابتدا صفر هست

۱۳ ابتدا همه بیت‌های f_0 همه‌ی بیت‌ها را یک می‌کنیم

predicate register در یک بود (در v_0 $X[i]$ است) این یک دستور شرطی است که می‌گه اگر not equal بود v_0 را یک کند (v_0 رجیستری است که در خط دوم enable کردیم) به Contor unit بگوید وقتی $v_4 - v_0$ می‌کنی به شرطی این کار را انجام بده که یک بیت هایش این رجیستر هم یک باشد و بعد هم store کنیم به آدرس v_4 که این به جاهایی store می‌شود که v_0 یک بود و بعد هم رجیستر را خاموش می‌کنیم هم عملیات وکتوری را خاموش می‌کنیم

اسلاید ۸۱۴

* SIMD extension

ما الان حدود ۳ تا مدل برای SIMD گفتیم. واقعیتی که وجود دارد SIMD عمدتاً روی media application ها کاربرد زیادی دارند مثل پردازش صورت تصویر. از آن طرف درایه‌های این وکتورها درایه‌های کوچکی هستند مثلاً پیکسل‌های عکس هر پیکسل یک ۸ بیتی است که از ۰ تا ۲۵۵ مراتب رنگ آن پیکسل را نشان دهد معمولاً هر یک درایه یک عکس یک ۶۴ بیتی لازم نیست و هر درایه این یک ۸ بیتی باشد ولی عملیات که قرار انجام شود روی اغلب داده‌ها می‌خواد انجام بشه.

Scanned by CamScanner

ما یک پردازنده Core i3 داریم که ۶۴ بیتی است (جمع کننده یا ضرب کننده ۶۴ بیتی یا ممکنه ۱۲۸ بیتی یا ۲۵۶ بیتی داشته باشه) وقتی یک جمع کننده صحیحاً ۶۴ بیتی داریم از اون طرف یک پردازنده عام منظوره اینتل تا الان پیش بینی میشه که یک تصویر را پردازش کنیم یک ماتریس و وکتوری برخورده پردازش اینجوری باشه. چرا باید از قابلیت معماری که تحت SIMD استفاده کنیم یعنی اگر یک جمع کننده ۶۴ بیتی داریم که ۸ بیت آن را می‌خواهیم $\frac{1}{8}$ از آن ۶۴ بیت را روشن کنیم و $\frac{7}{8}$ آن بیکار است پس می‌توان جمع کننده ۶۴ بیتی عملیات ۸ بیتی یا ۳۲ بیتی هم انجام بده ولی خب آلات قابل داریم SIMD Extension. می‌شه که خاصیت این جمع کننده ۶۴ بیتی را از اری بیت ۸ و ۱۶ و ۳۲ و ۴۰ و ۴۸ و ۵۶ (ضرایب ۸) carry chain را قطع کنیم (Cut که در FA ها از یک طبقه به یک طبقه دیگه می‌رفت) و خاصیت که این جمع کننده ۶۴ بیتی را به ۸ تا جمع کننده ۸ بیتی تقسیم کنیم آن وقت جمع کننده این قابلیت را دارد که مثل یک vector processor ۸ تا جمع همزمان را با یک عملیات و یک دستور جمع انجام بده این می‌شه SIMD extension. می‌توانیم پردازنده را وادار کنیم که اگر ۸ تا داده‌ای ۸ بیتی را می‌خوای جمع کنی با ۸ تا دستور جمع این کار را انجام نده با یک تک دستور جمع هر ۸ تا دستور جمع را انجام بده اما وقتی بخواهیم پردازنده اینتل را وادار به استفاده کنیم که این قابلیت در آن وجود داشته باشد به اندازه‌ی یک vector processor که optimize شده برای همین عملیات می‌خوب نمی‌تونه عمل بکنه.

یک مسئله‌ای که وجود داره اینه که تعداد داده‌هایی که می‌توانیم در پردازنده اینتل به صورت موازی process کنیم خیلی محدود و کمتر از چند ده تا است از یک vector processor.

از اون طرف ما می‌توانیم حلقه‌های آرایه‌ای دهی حلقه پیچیده‌ای استفاده کنیم توی پردازنده‌های اینتل وقتی می‌خواهیم درصد SIMD عمل کند و از اون طرف چیزی به اسم mask register داریم حتی vector length register هم وجود ندارد بنابراین یکم دست ما را می‌بندد اما این است در برنامه به جایی برخوردیم که بتوانیم به صورت SIMD اجرا کنیم را این قابلیت اینتل را استفاده کنیم و این را باید بسپریم به کامپایلر در برنامه نویس هیچ دخالتی در آن ندارد

اسلاید ۱۵

در سال ۱۹۹۶، intel MMX طرح جدید که این قابلیت را داشت که ۸ integer operation ۴ integer operation ۱۶ بیتی را به صورت دستورات SIMD برای ما انجام بده. این تغییر یک عملیات را یک FU ۶۴ بیتی داشته این پردازنده ما الان جمع کننده ۶۴ بیتی یا ضرب کننده ۶۴ بیتی و در ISA

Scanned by CamScanner

آن تعریف شده بود که این ضرب کننده یا جمع کننده این قابلیت را داره که به جای این که همیشه دو تا ۶۴ بیتی را جمع کنه که از
این طرف دیتاهای ما هم عموماً ۶۴ بیتی نیستند و عموماً integer و ۳۲ بیتی هستند و Month و Day و چهار تا ۳۲ بیتی های
۸ بیتی و ۱۶ بیتی هستند. این قابلیت وجود داره که بتوانیم از این قابلیت های سخت افزاری به صورت SIMD
استفاده کنیم و این را به کاصطلاح پیلیس می گفتند

- سال ۱۹۹۹، SIMD extention دستی طرح شد که ۸ تا integer operation، ۱۶ بیتی به ۴ تا
integer operation ۳۲ بیتی یا ۴ تا Floating point، ۳۲ بیتی را انجام بده یا این که
دو تا integer Floating point operation، ۶۴ بیتی را هم می توانست بیاورد که
این معنیش اینه که پردازنده ای که سال ۱۹۹۹ اصل طرح می کرد Fu، ۱۲۸ بیتی داشته و این Fu، ۱۲۸
بیتی می تونه ۴ تکه یا ۲ تکه بشه و روی n تا دیتا کارش را انجام بده

- سال ۲۰۰۸ Advanced vector Extensions طرح شده که می تونست ۴ تا Fp یا
integer operation، ۶۴ بیتی را همزمان انجام بده که معنیش اینه که اونجا Fu، ۲۵۶ بیتی وجود داشته
- در سال ۲۰۱۷ معماری که طراح شده این است که ۸ تا integer یا Fp، ۶۴ بیتی را می تونه به صورت
SIMD اجرا کنه اگر همین طور پیش برهم می تونیم بگیم که این پردازنده یک Fu، ۵۱۲ بیتی داره (آیا مانعی جلوش نیست
۵۱۲ بیتی داریم که می تونه ۸ تکه تقسیم بشه و ۸ تا جمع انجام بده یا پردازنده این قابلیت را داره که ۸ تا جمع ۶۴ بیتی
انجام بده که شبیه ۵۱۲ تا جمع)

* - پردازنده ی اصل اگر بخواهد از قابلیت SIMD بودن استفاده شود و از این extension هایی که گفتیم
بخواهیم استفاده کنیم حتماً باید دیتاهایی که قرار است به صورت همزمان پردازش شوند پشت سر هم باشند و به
صورت aligned شده در آرایه باشند (aligned یعنی اگر می خواهیم ۸ تا ۶۴ بیتی باید بتونیم همزمان با هم
بیان حتماً شروع دیتاها باید مضربی از ۸ باشد مثلاً مضربی از ۸ باشد قرار بگیره، شروع این vector
که می خواد n تا دیتا رو با هم process شوند نمی تواند هر جایی از حافظه باشد و حتماً باید مضربی از ۸ باشد)
دیدیم اول lect-chap5-A-7

* همانطور که می دانیم Thread level parellesim داشتن یعنی بتونیم از یک مسیر از اجرای دستورات داریم،
یعنی n تا دستور را به صورت موازی داریم اجرا می کنیم ولی حتماً باید از n واحد پردازشی که از n تا داشتیم

Scanned by CamScanner

اینجا یک مسیر مستقل از دستورات داخل تسک سر هم اجرا میشن که در هر مسیر تمام آن چیزی که در فصل ۱۲
گفتیم برای آن صدق می‌کند (هازارد ها، وابستگی ها) حالا اینجا بین thread ها وابستگی نمیتونه وجود داشته باشه

Year:..... Month:..... Day:.....

- بحث ما در thread level paralism معطوف است به ساختار MIMD است
- بحث‌مون را محدود می‌کنیم به ساختار multi process ها یا که shared memory به صورت tightly-coupled دارن

در thread level paralism اگرچه multi threading روی یک single processor هم موضوعیت دارد ولی از concept های thread level paralism computer Architecture وقتی است که multi core هم داشته باشیم. n تا واحد پردازشی داریم که این قابلیت را به ما میدن که process استفاده کنیم multiple thread و این mutiple thread ها را به صورت parallel روی muti processor اجرا کنیم

- پس اگر n تا واحد پردازشی را داشته باشیم برای این که بهره‌وری بالایی داشته باشیم باید به تعداد واحدهای پردازشی thread هم داشته باشیم. اگر thread ها بیشتر از تعداد core ها باشد تعدادی از واحدهای پردازشی بیکار می‌مانند. در واقعیت تعداد thread ها بیشتر از core های واحد پردازشی است

- اینکه هر thread چقدر کار را انجام می‌دهد می‌شه grain size یا اندازه بستگی به برای انجام فعالیت دارم

- thread ها می‌تواند برای این که data level parallism را برای ماشین‌های SIMD استفاده شون اما اگر بخواهیم برای data level parallism از multi threading استفاده کنیم overhead خیلی زیادی به ما تحمیل می‌شود. چون اگر بخواهیم از ساختار MIMD استفاده کنیم برای این که یک دستور SIMD را بتونیم اجرا کنیم بهره‌وری را داشته باشیم باید به تعداد operation ها باید instruction اجرا کنیم

در حالی که در DLP برای mutiple operation فقط یک instruction اجرا می‌کردیم

اسلاید ۵:

+ دسته‌ها muti core:

1) symetric muti processor (SMP) هستند. در این ساختار تعدادی core داریم که به یک حافظه مشترک است بین core ها که این حافظه مشترک یک uniform latency

Scanned by CamScanner

برای ما ایجاد می‌کند یک حافظه است که همه جا مشترک یکسان است مثل RAM که در سیستم‌مون داریم و در تعامل با multi core است که همه‌ی core های این RAM را یکسان می‌بینن

Year: Month: Day:

۲) Distributed shared memory (DSM). در این ساختار n تا واحد پردازشی داریم که هر واحد پردازشی برای خودش یک local memory داره به اضافه حافظه‌هایی که بقیه واحدهای پردازشی دارند و در فضای آدرس‌دهی هر پردازنده هم وجود دارد (همه پردازنده‌ها یک حافظه به صورت local دارند به اضافه n تا حافظه به صورت remote و تعدادی حافظه هامی تشکیل فضای آدرس‌دهی حافظه که همه‌ی پردازنده‌ها می‌بینند) اگر بیاد local باشه سریع‌تر access می‌شه تا در remote باشه)

جایی مثل ساختار NUMA (non-uniform memory access)

در ساختار DSM ارتباط بین واحدهای پردازشی از طریق interconection network است که می‌تواند به صورت point to point پردازنده‌ها را به هم وصل کند یعنی یک ساختار پیوسته باشد که ارتباط مستقیم بین دو تا core برقرار کند یا به صورت غیر مستقیم و توسط multi hop وصل کند (یعنی از یک جایی به جایی دیگر بپرد تا در نهایت به مقصد مورد نظر برسد)

از نظر معماری چه چیزهایی در thread level paralism داریم؟

ما از همه‌ی چیزهای یک core پردازشی تا 4 تا core نداشتیم، حافظه و cache مشترک هم داریم thread ها را داریم اجرا می‌کنیم، در سطح نرم‌افزار هم همه‌ی عملیاتی که ما انجام دادیم می‌خواد اجرا بشه، چه چیزهایی در سخت‌افزار وجود دارد و طراح processor باید مراقب چه چیزهایی باشد وقتی process ها یا core ها را به صورت موازی multi thread را اجرا کند؟

در چنین ساختاری می‌توانیم تا 4 process مستقل از هم را هر کدوم را روی core ها مون اجرا کنیم چه مشکلی پیش بیاید؟ اگر حافظه‌ها در خواست دادن باید به صورت عادلانه به شون دسترسی داده بشه. مهم‌ترین نگرانی اصلی ما از باب cache coherence است

اسلاید 36

cache coherence:

process ها هر کدوم برای خودشون task هاشون تعریف می‌شه و thread های مربوطشون روانی می‌شه، این وسط هر کدوم از core های ما یا هسته‌های پردازشی ما icache و dcache اختصاصی خودشون

Scanned by CamScanner

را دارند و یک level cache مشترک هم دارند. شرایطی هایی هست که thread های که روی processor های مختلف اجرا می‌شن یا مسیرهایی که به صورت موازی دارند می‌ران به هم داده‌هایی دیتای مشترک دارند. این

Year: Month: Day:

متفاوت و در سرور باشد برای یک ساختار multi core و یک مشکل پیدا کرده. باید این مسئله دیتای مشترک را حل کند

مثال: در dual core processor A و processor B هر دو دیتا دارم در خونه x حافظه را آورده و value آن ۰ است. processor A برای x داده خونه و کش آن miss می‌شه و باید دیتا را از خونه x می‌آره توی یک بلاک کش قرار می‌ده و مقدار 1 را به خودش می‌ده. از این دیتای x الان یه نسخه توی خونه حافظه است یه نسخه در کش پردازنده A است. processor B هم یک مدت می‌ره از همون خونه حافظه می‌خواد دیتا را بخونه. پس خونه x حافظه همون دیتا ~~processor~~ مشترک است بین processor A و processor B (بین thread هایی که بین A و B اجرا می‌شن مشترک است) و B هم آن را می‌خونه و در کش خودش قرار می‌ده پس x الان سه نسخه است. در cache ها چیزی read نمی‌کنیم باید دلش بخواد write کنه الان در مرحله ۳ هستیم می‌خوان توی خونه x حافظه write کنن. وقتی می‌خواد write کنه یا کش را در کش آپدیت می‌کنه. اگر سیاست ما write on back باشه پردازنده A بلاک کش را آپدیت می‌کنه ولی بلاک حافظه را آپدیت نمی‌کنه. اگر write through باشد هم در کش و هم در حافظه آپدیت می‌کند اما دیتای پردازنده B آپدیت نمی‌کنه. الان از x سه تا نسخه وجود داره که کلش با هم متفاوت است و اینجوری کل برنامه از سوال میره. cache coherence یعنی این که ما تضمین کنیم که این اتفاق نیفتد

اسلاید ۳۷

* - یعنی هر read توسط هر پردازنده از هر دیتایی همیشه تضمین شده باشد که most recently written value را برای هر پردازنده همیشه آپدیت باشیم

- مورد دوم که باید تضمین کند این است که write های که در یک خانه حافظه توسط processor های مختلف ممکن است انجام شود (یعنی هم A خواد عوض کند هم B) ترتیب write شدن توسط هر processor ها یک جور دیده شه

- coherence معمولاً در مقابل consistency قرار می‌گیرد (لازم و ملزوم هم هستند) consistency جنبه نرم‌افزاری و منطقی برنامه است که می‌گه یک دیتایی نوشته شده و وقتی می‌خوانیم value آن که به دست می‌رسد و نسخه درست آن چیه است (یعنی یک thread یه جایی می‌خواد read کند یعنی ترتیب write کنه این که این read و write چه ترتیبی نسبتی با هم داشته باشند می‌شه consistency

Scanned by CamScanner

ما باید یکسری دستورات کنترلی داریم که بگه اول کی بنویسه و چه کی بخونه و coherency هیچ کاری به این مورد نداره
فقط می‌گه که value جدید چی هست و الان باید چی را بخونه

Year: Month: Day:

یک مورد دیگه در consistency این است که اگر یک پردازنده در خونه A بنویسد باید توی خونه B بنویسد هر پردازنده
دیگه ای در سیستم، مقدار جدید B که توسط پردازنده A نوشته شده را وقتی می‌بینه قبلش باید نوشتن A را هم دیده باشه
(در پردازنده دیگه هم باید ببینه که ما اول A را آپدیت کردیم بعد B را) یعنی ترتیب آپدیت شدن را باید بدونن ولی
coherency کاری به این ترتیب ها نداره فقط مقدار نهایی مهم است)

اسلاید ۳۸

* cache coherence به مالکان migration و مهاجرت داده را می‌دهد بین core ها
یعنی بلاک کش را نمی‌ذاریم به یک پردازنده یا core دیگه رهو دون دیگه آن دیتا را نداشته باشیم به اضافه
این که امکان replication به ما می‌دهد (یعنی اگر یکی دیگه ای دیتا را خواست به اول هم بدیم و
multiple copies از آن دیتا داشته باشیم)

- برای این که coherency درست تضمین شود یکسری protocol وجود دارد که دو دسته هستند:
۱) Directory bace هستند که یکسری اطلاعات به خصوص status بلاک های کش حفظ را
در یک جای مشترک که همه ی core ها هستش ACess را ذخیره می‌کنیم و با استفاده از آن
جدول به اطلاعات مکان ذخیره شده می‌توانیم coherency را تضمین کنیم.

۲) snooping: اشتراک بهتر می‌گذارد bus که core ها ارتباط ایجاد کرده است، یعنی هر core یا
پردازنده ما گوش می‌دهد به bus که ارتباط برقرار کرده بین همه ی core ها و با استفاده از اطلاعاتی که روی bus
ادر بدل می‌شه سعی می‌کنه ان coherency را تضمین کند.

اسلاید ۳۹

* coherence protocol snoopy دو دسته هستند:

۱) write invalidate: هر وقت دیتایی در یکی از کش های core ها آپدیت می‌کنیم (در زمان write) کاری
که باید بکنیم آن بلاک در جاهای دیگه از سیستم اگر وجود دارد آن ها را invalidate کنیم (آنهایی که در کش های دیگه
هستند را invalidate کنیم و چیزی که باقی memory است باقی می‌شه invalidate نکرد)
در این سیاست باید تضمین کنیم که اگر می‌خواهیم دیتایی را write کنیم، قبل از زمانی که می‌خواهیم write کنیم و بعد

Scanned by CamScanner

به همه می‌گیم write کردم و نسخه‌ای که دست شما هست به‌روز نمی‌خواه، عکسه در یک بلوک بلافاصله بعد از این که من write
را انجام دادم کسی دیگه‌ای بخواد آن دیتا را read کند باید این را تضمین کنه cache coherency protocol

از اون لحظه‌ای که من write را در دیتا انجام دادم هیچ کس دیگه‌ای از نسخه‌های قبلی دیتا نباید استفاده کند. برای این که
این تضمین حاصل شود کسی که می‌خواد دیتایی را در کش آپدیت کند باید قبلش bus را تصرف کرده و بتونه این خبر
را روی ~~bus~~ bus بده که بقیه اگر در آستانه خواندن آن دیتا هستند دست نگه دارند (اگر در آستانه
نوشتن هم هستند همین طور) و اول اجازه بدن ما کارمون را انجام بدیم و بعد ادامه کارشون را انجام بدن

مثال در سیاست write invalidate

دو تا پردازنده A و B داریم، یک value صفر در خونه x حافظه نوشته شده. اول پردازنده A دستور می‌خونه
و میاره در کش، value آن صفر است. بعد پردازنده B آن را می‌خواد و کش آن هم miss می‌شه و دیتا را می‌آره
و در کش خودش قرار می‌ده. پس سه تا نسخه از x وجود داره. پردازنده A وقتی می‌خواد که در این خونه حافظه write
انجام بده bus را در اختیار می‌گیره و به همه خبر می‌ده که من این بلاک را آپدیت کردم و هر کی اگر دیتای x را
دارد آن invalidate کند. پس پردازنده B دیتای صفر را دور می‌زنه و از این به بعد آن دیتا را دیگه ندارد. حالا پردازنده B
اگر بخواد دوباره دیتا را بخواند miss می‌دهد پس می‌ره دیتای جدید را از حافظه می‌خواند و وقتی از یک به
دستش می‌رسد (در اینجا فرض بر سیاست write back است) پس چون دیتا در حافظه اصلی کثیف
شده پردازنده A باید حواسش باشه اگر کسی روی bus درخواست دیتایی را داد که من الان آپدیتش کردم باید
بلافاصله مانع از دسترسی آن پردازنده به main memory بشه و قبل از این که ریکوست پردازنده B به
memory برسه پردازنده A باید وارد عمل شود و بگه این دیتا که می‌خوای دست من است نزد از اول من بیام
در main memory نجوان تا main memory write back دیتا را main memory بگیرم بعد تو بردار
می‌تونی از همین bus که من دیتا را روش می‌نویسم که بره به main memory برسه خودت دیتا را بردار

۲) write update: زمانی که دیتایی را آپدیت می‌کنیم دیگه به بقیه نمی‌گیم دیتا را دور بریز، این
دیتای جدید هست اونی که داری بریز دور این دیتای جدید را برداره جایش. یعنی در مثال بالا پردازنده B وقتی دیتا را
دیگه دیتای جدید را جایش می‌ذاره پس لازم نیست این که پردازنده A وقتی دیتا را آپدیت کرد روی bus
هم بزاره (هر چی در کش می‌نویسه باید در bus هم بنویسه که خودت بقیه هم برسد)

Scanned by CamScanner

اسلاید ۱۰:

Year: Month: Day:

پس چیزی که نیستم اگر کش write back باشد هر دیتایی که آپدیت می‌شود باید اگر پردازنده دیتایی را درخواست را در کش آن وجود نداشت، باید کسی که دیتا را دارد حتماً حواسش به bus باشه و آپدیت دیتایی که دست خودش هست را بفرستد برای کسی که درخواست داده. این مسئله فقط snooping. همه‌ی کش‌ها دارن به bus گوش میدن و همه‌ی کش‌ها هر کدوم به ازای هر آدرسی که روی bus می‌افته باید بدن آن آدرس که روی bus هست را در کش خودشون سرچ کنند. یک پردازنده B نمیتونن مثلاً دیتا رو بخوانند miss شدم و این خیلی توسط پردازنده A باید پردازش بشه: یعنی بره دنبال آن آدرس در حافظه‌اش بگرده و اگر دید همچین دیتایی داره و در حالت dirty است بلافاصله دیتا را برای B بفرستد.

* هر بلاک کش در این پروتوکل سه state دارد. در سیاستی که ما الان صحبت می‌کنیم اگر بلاک ما valide باشد دیگر از دو تا state، shared (مشترک) و modifide یا exclusive است.

هر write که در یک بلاک کش انجام می‌شود اگر آن بلاک در state، shared باشد، آن processor که داره write انجام می‌ده و می‌بینه که داره write را روی بلاک shared انجام میده حتماً باید پیام invalidation را روی bus، broad cast کند یعنی حتماً باید به همه خبر بده من دارم این بلاک را آپدیت می‌کنم. پس یک بلاک کش سه تا state می‌تونه داشته باشد: یک بلاک کش یا valide است یا invalide (اگر invalide باشد یا valide است یا modified است (اگر عامل dirty می‌شد) یا shared است (که عامل clean می‌شود)

اسلاید ۱۱:

تمام اتفاقاتی که در یک بلاک کش ممکن است رخ دهد و عکس العمل کش را داده توضیح میده. این یک کش الان از دو طرف با دو تا چیز متفاوت در ارتباطه. از یک طرف با processor خودش (core خودش) در ارتباطه، اون core در هر بلاک چه چیزی محتوی رو گوش می‌ده و از بین دیتایی چه چیزی ارش دیتا بگیره. از اون طرف چون کش به یک bus وصله که داره ما نیز اتفاقاتی که روی bus اتفاق می‌افتد. این به چه صورت می‌شه چه اتفاقی اتفاقاتی می‌تونه در کش رخ بده که این اتفاقات یا مثلاً از سمت processor اومده باشه یا منشأ آن bus باشد پس آدرس‌ها از هر دو طرف به سمت ما روانه می‌شن و ما باید دنبالشون بگردیم چون چیزی که دنبال یک بلاک در کش می‌گردیم (توضیحات این اسلاید در سلام هست)

Scanned by CamScanner

اسلاید ۱۲:

Year:..... Month:..... Day:.....

این state machin برای توضیحات جدول اسلاید قبل هست.

هر بلاک یکی در یکی از این سه تا state هست و این transition ها اتفاقاتی است که در اسلاید قبل افتاده

مثلاً اون بالا بلاک read می‌تونه در invalid باشه می‌ره در state shared. در shared هر چند read hit رخ بده در همین state می‌مونیم اگر در shared باشیم و read miss رخ دهد دوباره در همین state می‌مونه ولی پیام read miss را باید روی bus بزاریم. اگر در shared write miss رخ بده در exclusive (modify) می‌ره و در همین state می‌مونیم و اگر write hit رخ بده در همین state می‌مونیم ولی حتماً باید پیام خودمون را روی bus بزاریم و وقتی write hit رخ میده پیام invalidation را باید روی bus بزاریم. اگر در state exclusive باشیم و مجبور باشیم این بلاک را replace کنیم با بلاک جدیدی که خاطر read miss وارد می‌شه باید این را در بیریم write back کنیم و بعد state آن را برگردونیم به shared و در exclusive هر چند که write hit و read hit شود در همین state می‌مانیم (این اتفاقاتی بود که از سمت cpu درش اتفاق می‌افتاد)

اتفاقاتی که از سمت bus درش می‌افتد می‌شه state machin سمت راست. خب در این حالت هیچ وقت توی state invalid چیزی از روی bus پیدا نمی‌کنیم و هر چیزی که روی bus به صورت پیدا می‌کنیم برای بلاک‌هایی است که وجود دارند یعنی دیتا در آن‌ها هست پس در state shared و state exclusive هستند و transition ها هم چون توضیحاتی است که در جدول داریم

* سوال: این پروتکل که به پروتکل MSI معروف است (MSI = modify, shared, invalid) آیا قدری به آن داریم چه؟ یا از نظر performance و بهره‌وری مطلوب عمل می‌کند؟

دوتا ایراد جدی وارده ← ۱) وقتی یک بلاک clean است این پروتکل می‌تونه تشخیص بده این بلاک clean متعلق به خودشه یا کسای دیگه هم این بلاک را دارند مخاطره چون وقتی می‌خواد یک بلاک clean را dirty کند نمی‌دانم از این که این بلاک بین کسی دیگه‌ای باشه یا نباشه این خبر را روی bus میذاره که من این بلاک را تغییر دادم هر کسی این بلاک را داره invalidate کنه ولی ممکنه هیچ کسی این بلاک را نداشته باشه و این یک کار بیهوده باشه

۲) فرض کنیم به عنوان یک core یک بلاکی را در state modified داریم یعنی آخرین نسخه نزد من است

Scanned by CamScanner

پس می‌دونیم هیچ core دیگه‌ای در کشش آن بلاک را ندارد و نسخه‌ای از این بلاک که در main memory
هست expired شده و به درد نمی‌خوره. core دیگه‌ای که درخواست این دیتا را می‌دهد

Year: ..... Month: ..... Day: .....

این پروتکل در این شرایط وقتی درخواست از سمت core دیگه‌ای می‌آید بلاک را write back می‌کند در
main memory و به آن core هم می‌دهد اما آیا این واقعاً لازمه این دیتا در main memory هم write back
بشه. main memory که چیزی نداره فقط سرعت کار خودش را می‌کند، یک core دیگه هم این بلاک
را می‌خواد. بلاک که دست ماهست آیا بهتر نیست بلاک را به کش دیگه بدیم کفایت می‌کند؟ چرا باید در main memory
هم write back کنیم. اگر بگیم چی می‌شه؟ اگر write back نکنیم پروتکل زندیسوال میره. بلاک ما
وقتی dirty است در exclusive است اگر این را به کش دیگه‌ای دادیم ما باید بلاک را در چه state
بدیم و این کش بلاک را باید در چه state بده وقتی بلاک dirty است. آیا state را عوض کردیم به shared
چون وقتی یک بلاک به core دیگه سپرده می‌شه state آن باید shared شود حتماً باید الان در
main memory، write back شود چون اگر نشود دیگه هیچ وقت نمی‌شه. اما این یک کار اضافه و بیهوده
است. ما کش write back را نداشتیم که مشکلات write through را نداشته باشیم و مجبور نباشیم
bus ها را هدر بدیم و این main memory به access داشته باشیم.

برای این که این دو تا مشکل حل شه هر دو تا اسلایدی دو تا پروتکل معرفی می‌شه که هر کدوم این مشکلات را حل می‌کنه

اسلاید ۱۳:

* MESI protocol

این پروتکل به بلاک‌های کش یک state اضافه می‌کنه که exclusive است. این state می‌گه که
فقط و فقط خودم بلاک را دارم. مهم نیست dirty یا clean است. مهم این که فقط خودم دارم و اگر
clean بود و بردم فقط خودم دارمش و بخوام dirty کنم دیگه نیاز نیست به کسی خبر بدم که این بلاک تغییر کرده
چون غیر من کسی این بلاک را ندارد و bus هم نیاز نیست اشغال بشه و خبری هم به بقیه کش‌ها دیگه
نمی‌دهم و هم bus اشغال نمی‌شه هم بقیه الکی درگیر نمی‌شن و باعث بهبود performance می‌شه چون هر بار
دیگه نباید درخواست‌های روی bus را مانیتور کنند و به ازای آن‌ها خودشان را درگیر کنند و از کارهای دیگه‌ای که CPU
باهاش داره جا بمونه و انرژی و پاورش هم بهبود پیدا می‌کنه.

۱۳۴ با توجه به راهنما مشخص که این ماشین چی کار می‌کنه. پس تفاوتش این که وقتی در exclusive هستیم اگر

Scanned by CamScanner

دیتا را write کردیم می‌ریم در modified و صحیح پیامی روی bus نمی‌دادیم، در shared اگر بودیم رفتیم write انجام بدیم میاییم در exclusive چون نمی‌دونیم که قبلی invalidate

کردند دیتا را، اگر در exclusive بودیم و روی bus کسی درخواست read نداد می‌ریم در shared اگر در modified بودیم و کسی درخواست read برای بلاک را بهش تحویل می‌دیم و خودمون هم write back می‌کنیم در main memory و بلاک را به بقیه هم می‌دیم. پس پروتکل MESI مشکل اول را حل می‌کند

اسلاید 31F

* مشکل دوم را با پروتکل MOESI حل می‌کنیم

این پروتکل یک state جدید به نام owned یا مالکیت بلاک ایجاد می‌کند. این state می‌گه زمانی که یک بلاک dirty داریم اجازه می‌دیم که بین کش‌های مختلف به اشتراک بذاشته بشه بدون این که در حافظه ذخیره بشه اما برای آن یک مالک تعیین می‌کنیم، آن مالک عضوی است که آخرین بار تغییر را در بلاک ایجاد کرده پس اگر بلاکی داریم که dirty و modified است اگر دیدیم که روی bus کسی پیام نداشت یعنی دیتا را خواست (پیام read) این بلاک را می‌دیم دادن کسی که می‌خواد ولی خودمون خودم را مسئول آن بلاک می‌دونم یعنی از آن به بعدی دونم غیر از من کش دیگه‌ای هم این بلاک را دارد اما آخرین نسخه درست من است و هر وقت این بلاک را خواستم دور بندازم حتماً write back می‌کنم در main memory

اگر روی modified باشیم و کسی ادعا کند این بلاک را می‌خوام write کنم آن وقت مجبوریم آن بلاک را invalidate کنیم و حتماً خواستیم بلاک جدید را از اون یکی بگیریم. بقیه مسائل هم مثل قبل است

ویدیو دوم (lec7_B_1)

اسلاید 34F

cache inclusion policy

ما قبلاً وقتی یک پرارزنده داشتیم یک cache L1 داشته باشیم، یک cache L2 و main memory هم داشته باشیم زمانی که پرارزنده درخواستی به L1 می‌ده، L1 اگر نداشته باشد درخواست را به L2 می‌ده و L2 هم اگر نداشته باشد به main memory می‌ده پس main memory به L2 می‌ده و L2 به L1 و L1 به پرارزنده می‌ده و L1 و L2 هم می‌دونند که چه چیزی در L1 و L2 هست. تا زمانی که از

Scanned by CamScanner

L2 بیرون انداخته می‌شه اگر dirty باشه باید برگرده به L2 اگر clean باشه کافیه حذف شه. مشکل اینه که در L2 جا نیست برای آن بلاک می‌اومده در L2 کنار آن بلاک replace شه با بلاک جدیدی و دیگه در L2 وجود نداشته باشه. اما مسئله‌ی محتوای L1 و L2 یه همین سادگی نیست و برای آنها هم سیاست‌هایی مطرح شده. سه تا سیاست در ارتباط با محتوای L1 و L2 وجود داره

1) inclusive policy

2) exclusive policy

3) Non-inclusive non exclusive (NINE)

اسلاید ۴

* - سیاست inclusive policy

در این سیاست هر آنچه که در cache L1 وجود داره باید در L2 هم باشد یعنی محتویات L1 همیشه زیرمجموعه‌ای از محتوای L2 باشه. فرض کنیم L1 و L2 را داریم. پردازنده درخواست داده‌ی x را می‌ده به L1 و L1 چون نداره به L2 می‌ده و L2 x را می‌خونه و نداره پس می‌ره از main memory می‌خونه و به L2 هم می‌ده و L2 به پردازنده می‌ده. پردازنده بعد از مدتی درخواست داده y را می‌ده، L1 نداره به L2 می‌ده، L2 هم نداره به main memory می‌ده و اون هم به L2 می‌ده. الان x و y در L1 و L2 هستند. بعد از مدتی در L1، x ها اخراج می‌شه و replace می‌شه با x جدید ولی x ها همچنان در L2 داریم، یک مدت می‌ذاره و y در L1 داره استفاده می‌شه و y در L2 اخراج می‌شه. سیاست inclusive می‌گه y نباید در L2 هم باشه پس y ها که در L1 داره استفاده می‌شه چون در L2 نیست طبق این سیاست دور می‌ریزیم. این سیاست به چه دردی می‌خوره؟

- سیاست Exclusive policy

می‌گه چرا ما هر چیزی که از main memory می‌آوریم به L2 می‌ریزیم چرا باید در حدی کش‌هایی که در مسیر است ذخیره شه. فکر ما چند بار می‌خوایم از آن داده استفاده کنیم؟ فضای کش‌ها محدود است باید حداکثر استفاده را از آن بکنیم. پس وقتی بلاکی را در L1 داریم تنها کسی که از آن داده می‌خواد استفاده کند پردازنده است که با L1 داره کار می‌کنه و اون بلاک در L2 و L3 هیچ کاربردی نداره پس چرا باید فضای L2 و L3 را اشغال کنیم، بیایم در L2 و L3 داده‌ای را بذاریم که در L1 نیست تا هر وقت خواست سریع بفرستیم برسونیم

Scanned by CamScanner

پس این سیاست می‌گه هر آنچه در L2 هست در L1 نباشد و هر آنچه در L2 هست در L1 هم نباید باشد پس اینجا کاملاً از هم باید جدا باشند. این سیاست برای این است که حداکثر بهره‌وری را داشته باشیم. اگر یک چیزی که از سمت main memory به L2 حواله می‌شه در مسیر ذخیره نشه پس L1 ما چه جوری باید پر بشه و data در خودش ذخیره نکنه. فرض کنیم پردازنده‌ای دارم با کش‌های خالی، پردازنده درخواست داده‌ی x را می‌ده، داده x در L1 نیست، L1 به L2 می‌گه x را داری، L2 به main memory می‌گه x را به ما بده، main memory، x را می‌ده به L2، L2 بدون این که x را در خودش ذخیره کنه می‌فرسته برای L1 و L1 آن را داره و می‌ده به پردازنده. پردازنده درخواست داده‌ی جدید y را می‌ده به L1، L1 هم نداره و به L2 می‌گه صبر کن برم از main memory بیارم، میاره و به L1 می‌ده ولی در L2 ذخیره نمی‌کنه. حالا L1 ما خالیه چه جوری پر می‌شه؟ از سمت L2 پر می‌شه. بعد ما چیزهایی را در L2 ذخیره می‌کنیم که L1 چیزهایی نداره در خودش یعنی هر آنچه L1 دارد در مسیر برگشت L1 الا به جای این که از سمت main memory باشه از سمت L2 است. پس هر چیزی که از main memory می‌آید مستقیم به L2 می‌ره و هر چیزی که از L2 بیرون انداخته می‌شه وارد L3 می‌شه چون می‌گه شاید در آینده نزدیک L1 دوباره آنها را بخواد (L1 چون پر شده مجبور شده آن بلاک را بیندازه بیرون پس L1 در خودش ذخیره می‌کنه شاید در آینده L1 دوباره آن را بخواد). برخلاف چیزی که در رابطه با cache writeback گفتیم در سیاست exclusive اتفاق می‌افتد این است که برخلاف آن چیزی که قبلاً در مورد cache writeback می‌گفتیم فقط بلاک‌هایی که dirty هستند باید برگردند main memory (به کش بعدی یعنی L2) می‌رن، الان هر چیزی که از L1 اخراج می‌شه باید write back بشه به L2 (تا الان می‌گفتیم هر چیزی که dirty باشه write back می‌شه به L2 و اگر clean باشه write back نمی‌شه ولی الان چه clean باشه چه dirty باشه می‌آید در L2)

- non-inclusive non-exclusive

در این سیاست کش L1 و L2 ما می‌گن به ما ربطی نداره در آن کش چه چیزی وجود داره، چیزی که من دارم برام مهم نیست اون کش داره یا نه و چیزی هم که ندارم برام مهم نیست اون کش داره یا نه. پس هر کش کار خودش را داره پس هر کش اگر چیزی نداره درخواست می‌ده که برایش بیاره و اگر هر چیزی را که می‌خواد دور می‌اندازه یا replace می‌کنه و بقیه کاری به اون ندارن و اون هم به بقیه کاری نداره. فرض کنیم پردازنده درخواست x را می‌ده L1 می‌ره و x را برای خودش پیدا می‌کنه، L1 به L2 می‌گه x را بده و L2 هم از main memory، x را می‌آره

Scanned by CamScanner

در خودش ذخیره می‌کنه و به P1 هم در خودش ذخیره می‌کنه و به P2 ارائه می‌ده، بلاک از حافظه توی x رکوشت می‌شه
و x در P1 و P2 وضعیت داخل main memory می‌آید و در P1 و P2 قرار می‌گیره و هر وقت می‌خوای main memory
می‌آید در خودش ذخیره می‌کنه) بلاک از حافظه x از P1 اخراج می‌شه، وقتی x از P1 خارج می‌شه، P1 هیچ
کاری نداره ولی من زمانی x را می‌اندازم بیرون که بخوام بلاکی را با x جایگزین کنم، بلاک از حافظه x از P1
اخراج می‌شه ولی برای x در P1 اتفاقی نمی‌افته

در cache coherency های جلسه گذشته دیدیم بدون این که بلاکی به کش وارد شه یک بلاک از
کش دور می‌اندازیم. این سازمان های invalidate و دور انداختن یک بلاک از کش منجر به این می‌شه
که بلاکی در کش وجود داره شاید بسیار مهم برای استفاده باشه ولی یک کش دیگه متعلق به هسته دیگه‌ای در
multi core ما آن کش را مجبور می‌کنه که بلاک را دور بندازد و خیلی زود ما دوباره به اون بلاک نیاز داریم
و بلاک ما در کش miss می‌شه (بلاکی که با چیزی جایگزین نکردیم و کش دیگه‌ای ما را مجبور کرد آن را
دور بیاندازیم)

نکته: cold miss ← miss که چون هنوز بلاک گرم نشده و تازه اومده و تازه پیدا شده رخ داده
miss که بخاطر invalidation یک بلاک در cache coherency پروتکل رخ میدهد از کدام جنس
miss است؟ compulsory نیست چون بلاک در کش بوده و دفعه اولش نیست که اومده، conflict
هم نیست چون کسی نیومده بگه من می‌خوام اینجا بشینم تو برو بیرون، capacity هم نیست چون
سایز کش ما که کوچکتر نبوده و اتفاقا سایزش را بزرگتر می‌کردیم باز هم اخراج می‌شد پس این miss
از یک نوع چهارم miss است که همون coherance miss گویند چون بخاطر cache coherancy
رخ داده (این یک miss دیگه به miss های ما اضافه می‌شه) غیر از اون در بحث inclusion policy
که در کش وجود دارد گاهی وقت‌ها به ما دیکته می‌کنه که بلاکی را از کش دور بندازیم زمانی بود که در P1 و x
وجود دارند و اگر x از P1 اخراج می‌شد P1 را مجبور می‌کردیم دور بندازد و دفعه بعد که رکوشت بهش miss می‌شه
این نوع miss چیه؟ دیگه بخاطر coherancy نیست که سم cache coherancy است

استاندارد:

* Directory protocols:

در snoopling دیدیم که یک Bus داریم که همه‌ی core ها با هم وصل کرده و هر اتفاقی که در bus می‌افته

Scanned by CamScanner

باید روی bus مخابره بشه و همه‌ی cache ها باید bus را رصد کنند و در خودشون هم سرچ کنند و اقدامات لازم را انجام دهند این چیه مشکلی ایجاد می‌کنه؟

Year:..... Month:..... Day:.....

۱) scalable نیست. هر چقدر که تعداد core ها زیاد شن بیشتر کنسم همه‌ی ترافیک و کارهای انجام شده برای کش‌ها و bus ها به خاطر coherency بیشتر می‌شه. اگر تعداد core ها را از ۴ بکنیم ۴ تعداد ترافیک‌ها هم روی bus چند برابر می‌شه. bus هم چقدر ظرفیت داره. آن کش که واقعیش اینه که core خودش سرویس بده آیا اگر تعداد core های کناری زیاد شن اصلاً فرصت می‌کنه core خودش سرویس بده یا همش باید پیام‌های روی bus را رصد کنه و سرچ کنه که چیکار باید کنه. scalable نیست چون از یه حدی تعداد core ها را از یه تعدادی نمی‌توانیم بالا ببریم (۲ و ۴ تا core می‌تونه باشه و الا باید همش اتفاقات روی bus را رصد کنه)

راه حل این که به جای این که همه‌ی cache ها در تراکنش‌ها روی bus باشند distributed باشند (مدیریت به صورت) که هر کش خودش، خودش را مدیریت کند، یک مدیریت centralized یا متمرکز داشته باشیم به این صورت که یک directory داشته باشیم در یک level بالاتر از این cache های که با هم communication دارند با این همه‌ی اطلاعات مربوط به بلاک‌هایی که در این کش‌ها هستند در directory در کش بالاتر وجود داشته باشه و در این دایرکتوری می‌دونیم چه بلاک‌هایی از حافظه در چه کش‌هایی هست و در این کش‌ها در چه state هست پس اگر فرض کنیم ۱ تا core داریم که هر کدوم کش local خودش را داره یک کش مشترک بالاسرشون هست مثلاً اگر سه level کش داریم هر core کش L1 و L2 خودش را داره که اختصاصی است و یک L3 به صورت مشترک. این L3 چیکار می‌کنه برای این که بحث coherency را مدیریت کنه. مادام L3 به ازای هر بلاک کش یک vector ذخیره می‌کنه (یک چندبعدی ذخیره می‌کنیم) که این چندبعدی به ازای تعداد core هایی که ما در پردازنده داریم اطلاعات ذخیره می‌کنه، هر بلاک L3 می‌تونه به L2 هر کدوم از core ها داده شده و L3 این اطلاعات را در ازای بلاکی که داره ذخیره می‌کنه روی دو تا این بلاک‌ها core 1 و core 3 داده هر کدوم اون کش بلاک را به core 2 و core 4 و ... داده و این اطلاعات را برای تک تک بلاک‌ها در خودش ذخیره می‌کنه. اگر فرض کنیم تمام اطلاعات مربوط به تمام بلاک‌ها که در دست تمام کش‌های local ما هست در L3 قرار است نگه‌داری شود آیا این را می‌پذیریم که کش L3 ما حتماً نسبت به L2 ها باید حالت inclusive داشته باشد. پس L3 یک super set است از کل کش‌های L2 ها که این همون سیاست inclusive است

Scanned by CamScanner

اسلاید ۳۶ *

یک حالت دیگه در مورد ساختار یک سیستم ماشین موازی این بود که در کنار SMP (shared multi processor)

Year:..... Month:..... Day:.....

می‌توسیم حالت distributed shared memory را در نظر بگیریم که ساختارش را اینجا می‌بینیم

n تا پردازنده به همراه کش‌ها و local memory ها از طریق یک بستر ارتباطی به هم متصل باشند و همه واحد

پردازشی ما فضای حافظه‌ای کلی بین اجتماع تمام حافظه‌ها این است که اینجا وجود داره. در اینجا همه‌ی این memory ها

می‌شن فضای کل حافظه سیستم که همه بهش Access دارن البته Access به صورت NoMA هست به این معنی که

هر Core دسترسی به حافظه local خودش سریع‌تر است از دسترسی به حافظه‌های remote. در این ساختار می‌توانیم

ماشینی با این همه‌ی این واحدهای پردازشی انجام بدیم یعنی یک بلاک حافظه ممکنه در کش‌های processor های

مختلف وجود باشه این که اطلاعات مربوط به این بلاک‌ها فقط ذخیره نشده می‌تواند در یک directory های باشد که

هر بود برای خودش داره. این دایرکتوری‌ها به صورت distributed نظارت می‌کنند بر کل بلاک‌های

همه‌ی حافظه‌ها به این صورت که این به ازای یک یک بلاک‌هایی که در memory هست و می‌تواند

وارد کش‌های مختلف بشوند این دایرکتوری اطلاعاتی که تقسیم را ذخیره می‌کنه # در دایرکتوری این اطلاعات وجود داره

که این بلاک در کدوم کش هست و در چه وضعیتی است و هر وقت اطلاعاتی می‌خواد بنویسه روی بلاک مثلاً dean

هست و می‌خواد dirty شه باید این اطلاعات را به کوش بقیه‌ی پردازنده رسونه اقدامات لازم را انجام بدن و

خاصیت scalable می‌تونه اضافه بره و تعداد نودها را اگر بیشتر کنیم خوبی جای فراوانی نداره چون هر بلاک حافظه در کش

دایرکتوری خودش هست هر چند که پیام‌های مبادله شده با افزایش نودها بیشتر است ولی اگر خودت مدیریت

کنیم تقسیم وظایف بین Core ها و آن وقت می‌توسیم امیدوار باشیم که هر Core Access ها

به حافظه local memory خودش باشه و فقط برای مسیر مورد دیتاهایی از memory دیگران بخواد و

در این صورت می‌توسیم امیدوار باشیم در هر کشی عمدتاً بلاک‌های مربوط به حافظه local وجود داره و

ندرت وجود داره که دیتاهای مربوط به دیگران در این کش باشه در این صورت دایرکتوری فقط برای مسیر مورد

در بین این bus هست

اسلاید ۳۷

* پروتکل مربوط به دایرکتوری چیزی متفاوت از روشی که در snooping دیدیم نسبت فقط جای ذخیره شدن

اطلاعات و آن کسی که نظارت و مدیریت می‌کند متفاوت است در روش دایرکتوری با همون ساختاری

که تقسیم هر بلاک کش مثل پروتکل MSI در یکی از سه تا state می‌تونه باشه، ۱) shared به این معنی که

این بلاک memory در دست چند تا از process ها و کش‌های local هست و در state ،

Scanned by CamScanner

clean است و مادر دایرکتوری می‌گوید یک تک core هایی که این بلاک را دارند حتماً آن را ذخیره کردیم و در اختیار داریم پس می‌دانیم که این بلاک حداقل در دست چهار پردازنده یا local cache هایی هستند

Year:..... Month:..... Day:.....

۲) uncached: یعنی این بلاک memory دست هیچ core به صورت local نیست و هیچ کسی درخواستش را نداده و یا اگر داده دستش نیست

۳) modified: یعنی فقط یکی از کش های local ما این دیتایی که در main memory هست را در اختیار دارد دیتا را هم عوض کرده پس دیتایی که نزد مادر memory هست out of date است و دیگه نباید بدرد بخوره و باید هواسمون باشه دیتا را ازش پس بگیریم اگر خواست دور بزاره پس ما شماره یک core را فقط نگه داریم این بلاک memory یا در دایرکتوری مربوط به آن بلاک ذخیره می‌کنیم و آن دایرکتوری است که پیام های مربوط به invalidation و پیام های دیگه را باید بین پردازنده ها مخابره کند تا مدیریت را انجام دهد

اسلاید ۸:

جدولی مشابه چیزی که در snoping های cache coherancy protocol دیدیم در اینجا هم هست. اقدامات کدوم اجزا بین پردازنده، کش، دایرکتوری و دیگر core ها

اسلاید ۹:

پروتکل همون پروتکل MSI است. در کش به صورت local، هر کش هر بلاک را در یکی از سه تا state، invalide، shared و modified دارد هر کش برای خودش این state ها را نگه داری می‌کند از اون طرف دایرکتوری هم به ازای هر بلاک کش به جای هر بلاک که مربوط به اون memory هست یک state machin مثل بالا داشت نگه داری می‌کند، یک دو بیتی می‌گه که این بلاک آیا به کسی دادم یا نه، uncached یعنی به کسی ندادم، اگر به کسی دادم به چه کسانی دادم که می‌شه حالت shared، هر کسی که رفته گرفته بلاک مادر shared state بدونه به کدوم processor های دارنده آن بلاک اضافه می‌شه همین بلاک هم ممکنه در پردازنده های دیگه رخ بده که اون حالت oop یا در shared می‌شه، بلاکی نزد چند پردازنده داره که به صورت clean است، ممکنه یکی از پردازنده ها برایشون write miss رخ بده در این صورت باید بلاک را بدهم به حالت exclusive که فقط یک نفر به صورت انحصاری بلاک را در اختیار داره و به همه ی افراد دیگه بگید که بلاکشون را invalidate کنن و تنها دارنده آن بلاک بشود آن

Scanned by CamScanner

تک پردازنده‌ای که درخواست write miss داده، بلاکی در exclusive است یعنی فقط یک پردازنده
این بلاک را در اختیار دارد، یک پردازنده دیگه موقع میده که من این بلاک را می‌خوام write بکنم و
Year: Month: Day:
در من miss شده، باید فوری بلاک را از پردازنده می‌گیره و به این یکی می‌ده، این زمان fetch و
invalidate عمل می‌کنه و مالک بلاک را تغییر میده به پردازنده جدید. در حالت exclusive وقتی
که دیتا از کش یک پردازنده‌ای اخراج بشه write back بشه بدون این که کش دیگه‌ای آن را بخواد پس
write back می‌شه و برمی‌گرده به uncached و هیچ کشی آن بلاک را نداره

اسلاید ۱۰:

توضیحات این اسلاید هم همون توضیحاتی است که در شکل قبل دادیم
برای بلاکی که uncached و در کسی نیست:
- اگر read miss رخ بده بلاک را بهش می‌دیم و آن می‌شه تنها دارنده بلاک
- اگر write miss رخ بده بلاک را بهش تحویل می‌دیم اما از این به بعد بلاک به صورت انحصاری در اختیار
این core قرار می‌گیره و به شرطی به کش دیگه‌ای می‌دیم که از نو اول پس گرفته باشیم
برای بلاکی که در shared است:
- اگر read miss رخ بده، مثلاً چند نفر این بلاک را می‌خوان، این هم میادی که منتظر می‌خواد پس
بلاک را به این هم می‌دیم و این هم به لیست دارندگان اضافه می‌شه
- اگر write miss رخ بده باید تمام کپی‌هایی که این بلاک را دارند بلاکشون را invalidate کنن و
تنها دارنده بلاک به صورت exclusive بشود کسی که در آن write miss رخ داده

اسلاید ۱۱:

اگر بلاک exclusive باشه:
- اگر read miss از سمت کسی بیاد باید بلاک را از کسی که تا الان در اختیار داشته پس بگیریم (فقط ازش
بگیریم و مجبورش نکنیم invalidate کنه) و بلاک به state shared بره و به جای این که
به صورت انحصاری نزدیک نفر باشه، به صورت shared نزد دو نفر است
- اگر write back بشه یعنی از کش اخراج بشه بدون این که کسی آن را بخواد بلاک می‌شه uncached

Scanned by CamScanner

- اگر write miss رخ بده یعنی بلاک جدید را می‌آره بیاره و در آن بلاک جدید می‌آره write

انجام میشه پس این بلاک باید write back میشه و بلاک جدیدی که میاد در exclusive

Year:..... Month:..... Day:.....

Scanned by CamScanner