Data locality

По видео на Атанас Семерджиев на тема Data Locality

Ефектът му върху програмите и как най-добре да се възползваме от него

- *Как работи процесорът и как той достъпва части от паметта?*

(Обобщение от високо ниво като техническите детайли на всяка една архитектура варират или не присъстват всичките елементи)

Процесорът "живее" на дъното на нещо, което може да си представим като фуния. Първото нещо, което стои над него са нива на кеш памет (*n* на брой, като се означават с *L1, L2, L3..Ln*). След това стои *RAM* паметта, последвана от множество устройства, които също са вързани към машината (диск, мрежа, външни носители като *CD* и т.н.).

Ако има например променлива в паметта, до която процесорът трябва да достигне,то колкото по-надолу в йерархията се намира тази променлива, толкова по-бавно се случва това и се появяват все повече рискове за грешки.

За да се оптимизира този процес, кеш паметта наведнъж прави копия на нужната информация. Например, ако ни е нужна променливата *X*, която има размер *4B*, процесорът няма да прочете само тези *4B*, а вместо това той прочита данните, но и тяхната околност. Терминът за това е *cache line*. След това тази линия се появява като копие в *L1* (размерът ѝ е *64B*).

Колкото нависоко в йерархията на паметта се намираме, толкова по-малки стават обемите памет. Това е така по различни причини - кеш паметта в *L1* и *L2* е най-скъпа, а, за сравнение, дискът струва много по-малко. Второто съображение е, че има разлика и във физическия обем на частите и това носи със себе си проблеми, някои от които не са практически възможни за решаване.

За да не се натрупва памет, процесорът непрекъснато чете различни кеш линии от паметта, прехвърля ги в нивата на кеш памет и когато там свърши мястото, преценява коя кеш линия да изхвърли, за да сложи на нейно място ново прочетената. Поради тази причина, за да се постигне оптимизация, всеки производител има различен механизъм за съответната архитектура. Една разлика между архитектурите е например при прехвърляне на кеш линия от *L2* към *L1* - дали копието в *L2* се запазва, или се изтрива. Също така по различен се организират кеш линиите в нивата на кеш памет.

При работа с диска се използват функции например от фамилията *fopen/fread/fwrite/fclose.* След това някакво количество данни могат да бъдат прехвърлени от диска в *RAM* паметта, откъдето, в последствие, се работи с тях, и след обработка могат да бъдат върнати обратно в диска. При това прехвърляне минималният размер на блок памет е *4kB*. По правило операционната

система поддържа свои кеш линии, които правят копия на повече блокове памет от диска, за да спести ненужни прехвърляния. Тук една забележка е насочена относно функцията *CreateFile()*, която приема параметри (оказва се дали файлът ще бъде използван за четене/писане/четене и писане/добавяне само в края и т.н.). След конкретизиране системата може да оптимизира начина, по който кешира.

Note: изразът "прозрачен" в IT света: често се има предвид "скрит"

Този механизъм с кеш линиите обикновено е направен така, че да изглежда прозрачен за програмиста - адаптира се сам, за да стане процесът по-оптимален и да няма нужда от намеса, тъй като има прекалено много детайли около него.

- *Spacial Locality*

Когато две неща са много близо в паметта и ги използваме, това е оптималният вариант. Докато се работи с първото, е много вероятно вече да е кеширано и второто. За това, в някакъв смисъл, е за предпочитане работата с неща, които се намират близо в паметта (това невинаги е вярно!).

- *Temporal Locality*

Ако бъде кеширана една линия, има вероятност програмата дълго време да няма нужда от тези данни. През това време работят други процеси и се случва така, че дадената кеш линия бива свалена от паметта и на нейно място се кешира друга. Дори след време да бъдат нужни данните, те вече са изхвърлени. Затова е препоръчително достъпът до тези данни да се случва в по-кратък период от време (той се определя относително).

- *Примери*

По правило масивите имат "добро" *locality*, а свързаният списък - не толкова "добро". Също като списъка дърветата имат "лошо" *locality*. При такива структури от данни предимствата са други - например добър клас на времева сложност.

Класически пример за това как най-добре може да се възползваме от особеностите на този механизъм, е двумерният масив. В паметта той се представя като един непрекъснат буфер, в който са наредени редовете от таблицата, която ние си представяме всъщност (тоест всичко е наредено последователно по редове). Тогава, докато обходим първата колона и стигнем до първия елемент на втората колона, би могло да се наложи отново да се вземе информацията от паметта. Стига масивът да е достатъчно голям, всяко четене по колона ще е свързано с това да се отиде до паметта. Тя е била кеширана при първото четене от този ред (отделена е кеш линия с размер *64B*, която е съдържала елементи от първия ред), но поради големия размер, кеш

линията вече е свалена, за да бъде заменена от друга. Затова в много от случаите е по-оптимално четенето по редове.

- *Заключение*

Понякога, за да се съобразим с *Data Locality* се налага да се пише нечетим код или да се промени много логиката на програмата (не се препоръчва). Една добра практика е при писане на код да се има предвид, че *Data Locality* съществува и какви са ефектите от него. Съобразено с това да се оптимизира работата в ситуациите, в които това е възможно.