

Кеш меморија

Напредне теме



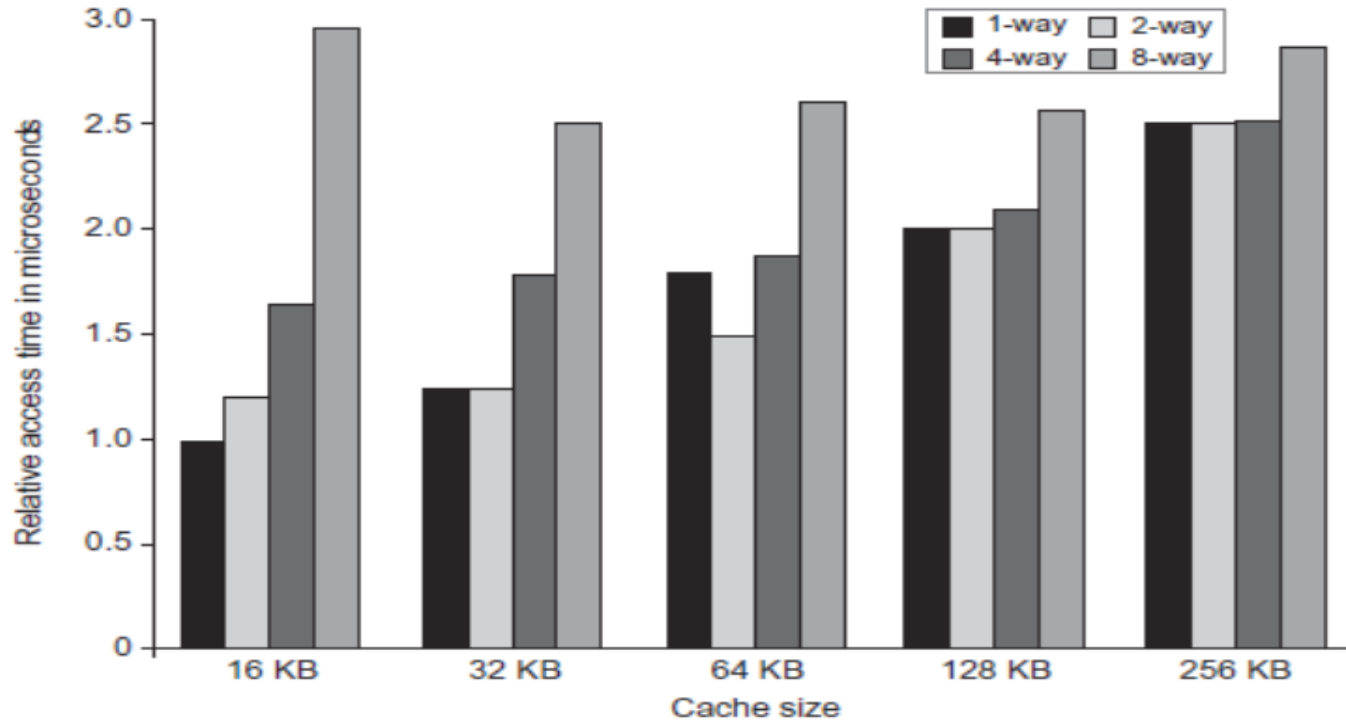
Садржај

- **Напредне технике оптимизације**
 - Хардверске технике
 - Софтверске технике
 - Наменске инструкције
- **Алгоритми замене**
 - Приближни LRU алгоритам
 - Побољшани LRU алгоритам
 - Остали алгоритми

Advanced Optimizations

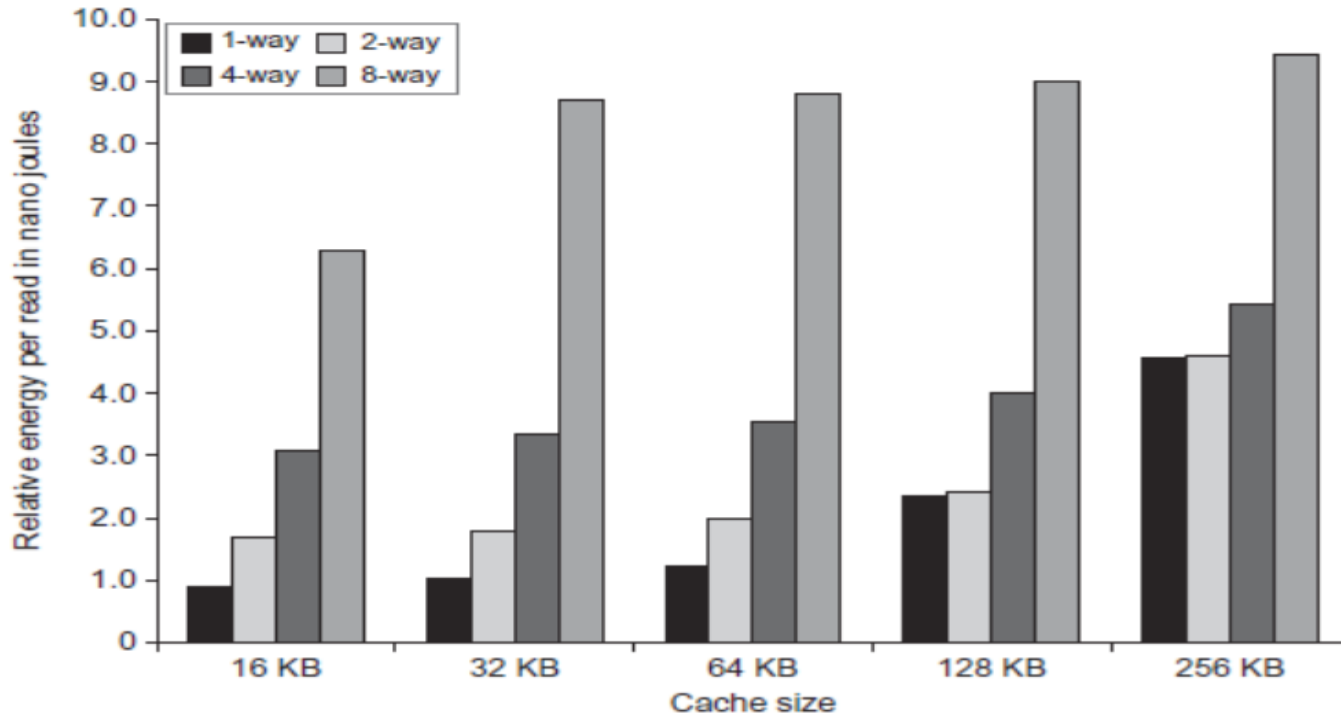
- Reduce hit time
 - Small and simple first-level caches
 - Way prediction
- Increase bandwidth
 - Pipelined caches, multibanked caches, non-blocking caches
- Reduce miss penalty
 - Critical word first, merging write buffers
- Reduce miss rate
 - Compiler optimizations
- Reduce miss penalty or miss rate via parallelization
 - Hardware or compiler prefetching

L1 Size and Associativity



Access time vs. size and associativity

L1 Size and Associativity



Energy per read vs. size and associativity

Way Prediction

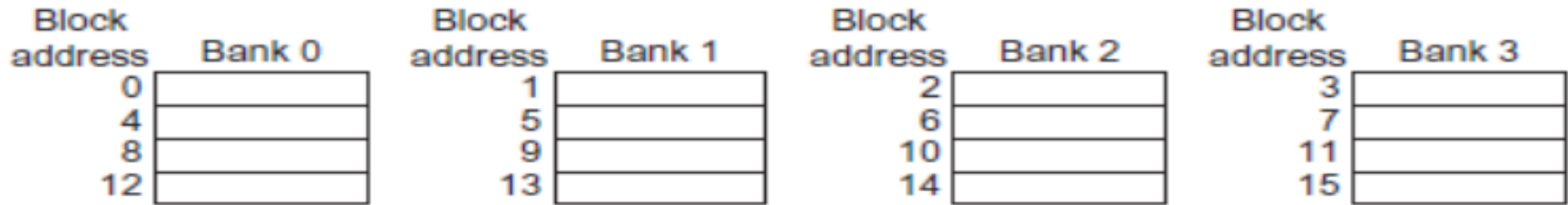
- To improve hit time, predict the way to pre-set mux
 - Mis-prediction gives longer hit time
 - Prediction accuracy
 - > 90% for two-way
 - > 80% for four-way
 - I-cache has better accuracy than D-cache
 - First used on MIPS R10000 in mid-90s
 - Used on ARM Cortex-A8
- Extend to predict block as well
 - “Way selection”
 - Increases mis-prediction penalty

Pipelined Caches

- Pipeline cache access to improve bandwidth
 - Examples:
 - Pentium: 1 cycle
 - Pentium Pro – Pentium III: 2 cycles
 - Pentium 4 – Core i7: 4 cycles
- Increases branch mis-prediction penalty
- Makes it easier to increase associativity

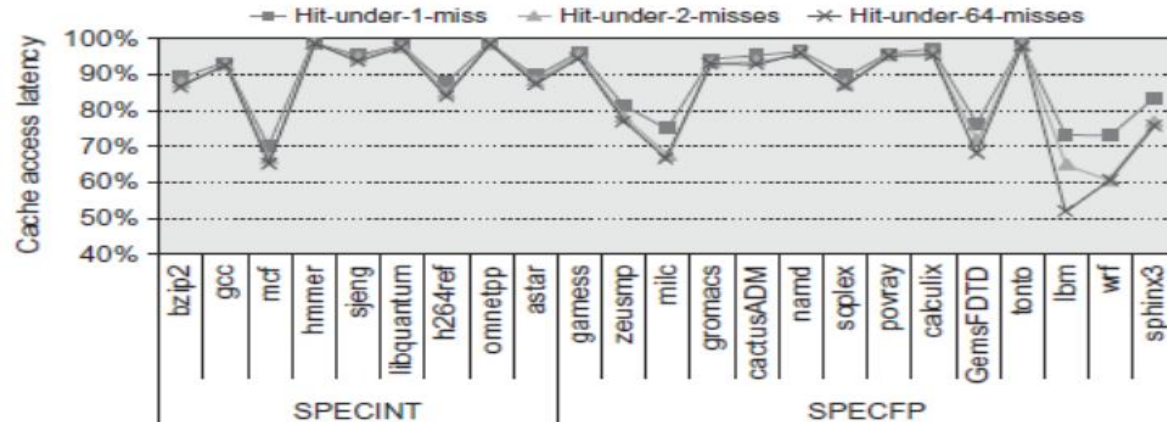
Multibanked Caches

- Organize cache as independent banks to support simultaneous access
 - ARM Cortex-A8 supports 1-4 banks for L2
 - Intel i7 supports 4 banks for L1 and 8 banks for L2
- Interleave banks according to block address



Nonblocking Caches

- Allow hits before previous misses complete
 - “Hit under miss”
 - “Hit under multiple miss”
- L2 must support this
- In general, processors can hide L1 miss penalty but not L2 miss penalty



Critical Word First, Early Restart

- Critical word first
 - Request missed word from memory first
 - Send it to the processor as soon as it arrives
- Early restart
 - Request words in normal order
 - Send missed work to the processor as soon as it arrives
- Effectiveness of these strategies depends on block size and likelihood of another access to the portion of the block that has not yet been fetched

Merging Write Buffer

- When storing to a block that is already pending in the write buffer, update write buffer
- Reduces stalls due to full write buffer
- Do not apply to I/O addresses

Write address	V	V	V	V
100	1	Mem[100]	0	0
108	1	Mem[108]	0	0
116	1	Mem[116]	0	0
124	1	Mem[124]	0	0

No write
buffering

Write address	V	V	V	V
100	1	Mem[100]	1	Mem[108]
	0		0	0
	0		0	0
	0		0	0

Write buffering

Reduce misses by compiler optimizations

- Instructions
 - Reorder procedures in memory so as to reduce conflict misses
 - Profiling to look at conflicts(using tools they developed)
- Data
 - *Merging Arrays: improve spatial locality by single array of compound elements vs. 2 arrays*
 - *Loop Interchange: change nesting of loops to access data in order stored in memory*
 - *Loop Fusion: Combine 2 independent loops that have same looping and some variables overlap*
 - *Blocking: Improve temporal locality by accessing “blocks” of data repeatedly vs. going down whole columns or rows*

McFarling [1989] reduced caches misses by 75% on 8KB direct mapped cache, 4 byte blocks in software

Merging arrays example

/* Before: 2 sequential arrays */

int val[SIZE];

int key[SIZE];

/* After: 1 array of structures */

struct merge {

int val;

int key;

};

struct merge merged_array[SIZE];

Reducing conflicts between val & key -> improve spatial locality

Loop interchange example

/* Before */

```
for (k = 0; k < 100; k = k+1)
  for (j = 0; j < 100; j = j+1)
    for (i = 0; i < 5000; i = i+1)
      x[i][j] = 2 * x[i][j];
```

/* After */

```
for (k = 0; k < 100; k = k+1)
  for (i = 0; i < 5000; i = i+1)
    for (j = 0; j < 100; j = j+1)
      x[i][j] = 2 * x[i][j];
```

- Sequential accesses instead of striding through memory every 100 words; improved spatial locality

Loop fusion example

/* Before */

```
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        d[i][j] = a[i][j] + c[i][j];
```

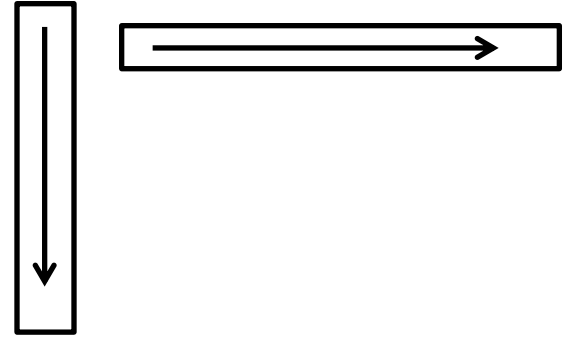
/* After */

```
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1){
        a[i][j] = 1/b[i][j] * c[i][j];
        d[i][j] = a[i][j] + c[i][j];
    }
```

2 misses per access to a & c vs. one miss per access; improve spatial locality

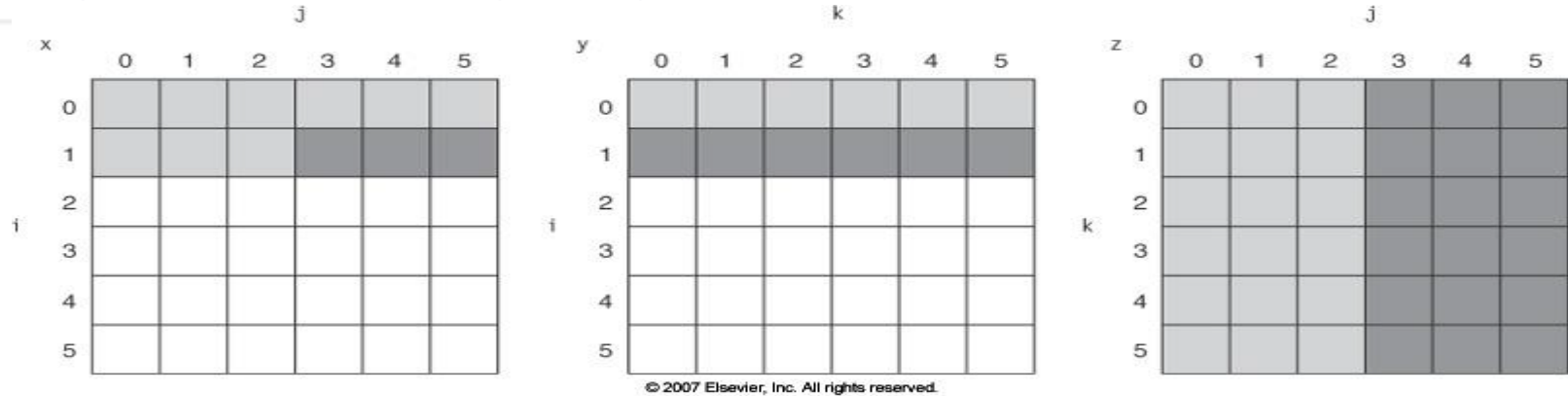
Blocking example

```
/* Before */  
for (i = 0; i < N; i = i+1)  
    for (j = 0; j < N; j = j+1){  
        r = 0;  
        for (k = 0; k < N; k = k+1){  
            r = r + y[i][k]*z[k][j];  
            x[i][j] = r;  
        };  
    };
```



- Two Inner Loops:
 - Read all $N \times N$ elements of $z[]$
 - Read N elements of 1 row of $y[]$ repeatedly
 - Write N elements of 1 row of $x[]$
 - Capacity Misses a function of N & Cache Size:
 - $2N^3 + N^2 \Rightarrow$ (assuming no conflict; otherwise ...)
 - Idea: compute on $B \times B$ submatrix that fits

Snapshot of arrays x,y,z for N=6 and i=1



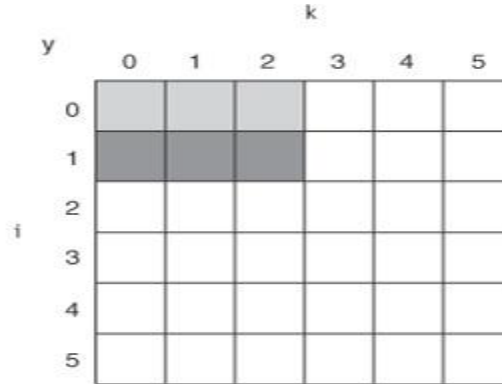
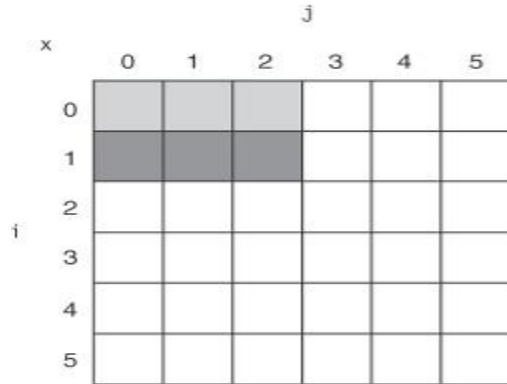
- The age of access to the array elements is indicated by shade.
 - White -> not yet touched
 - Light -> older access
 - Dark -> new access
- In the “before” algorithm the elements of y and z are read repeatedly to calculate x. Compare with the next slide which shows the “after” access patterns. Indexes, i, j, and k are shown along the rows and columns.

Blocking example

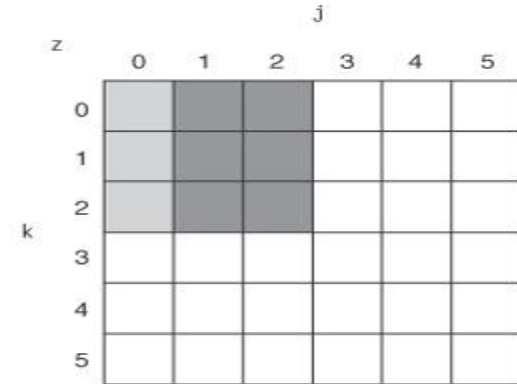
```
/* After */  
for (jj = 0; jj < N; jj = jj+B)  
    for (kk = 0; kk < N; kk = kk+B)  
        for (i = 0; i < N; i = i+1)  
            for (j = jj; j < min(jj+B-1,N); j = j+1){  
                r = 0;  
                for (k = kk; k < min(kk+B-1,N); k = k+1){  
                    r = r + y[i][k]*z[k][j];  
                }  
                x[i][j] = x[i][j] + r;  
            }  
        }  
    }  
};
```

- B called *Blocking Factor*
- Capacity Misses from $2N^3 + N^2$ to $2N^3/B + N^2$
- Conflict Misses Too?

Snapshot of arrays x,y,z for N=6 and i=1

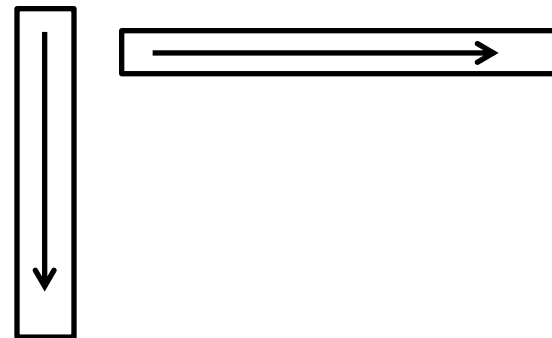


© 2007 Elsevier, Inc. All rights reserved.



Blocking example

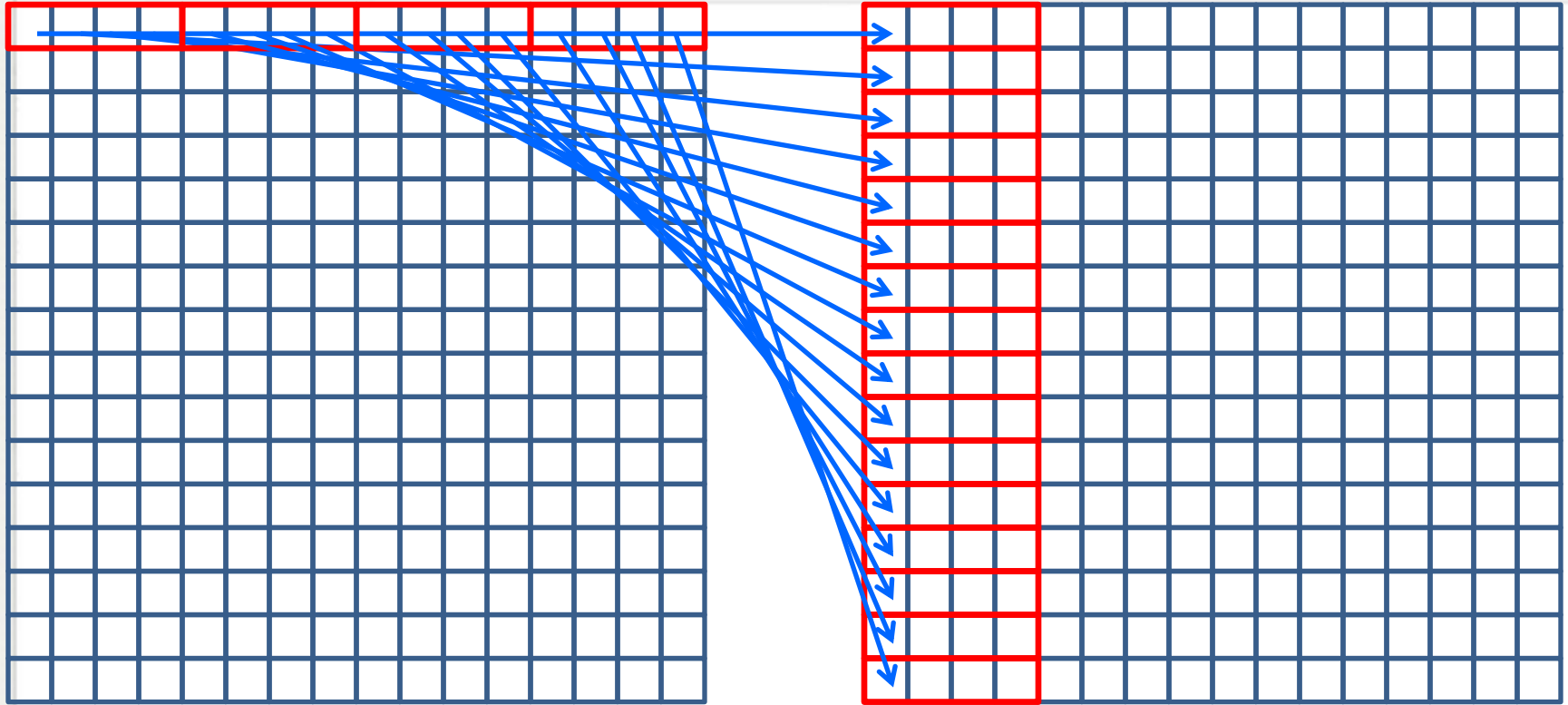
```
/* Before */  
for(int i = 0; i < N; i++){  
    for(int j = 0; j < N; j++){  
        a[j][i] = b[i][j];  
    }  
}
```



Two Loops:

- Read N elements of 1 row of $b[]$
- Write N elements of 1 column of $b[]$
- Capacity Misses a function of N & Cache Size:
- Idea: compute on $B \times B$ submatrix that fits

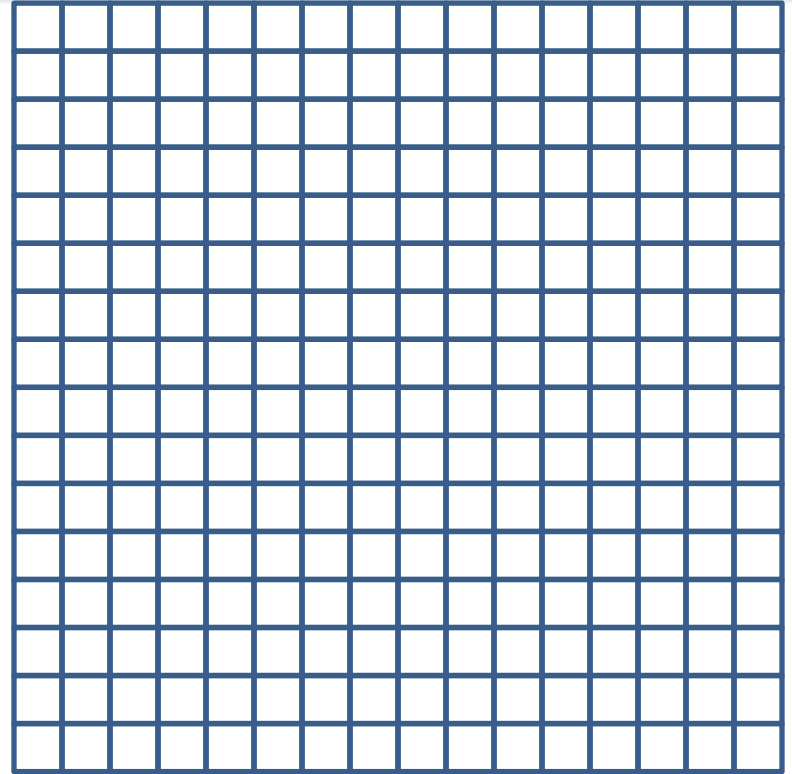
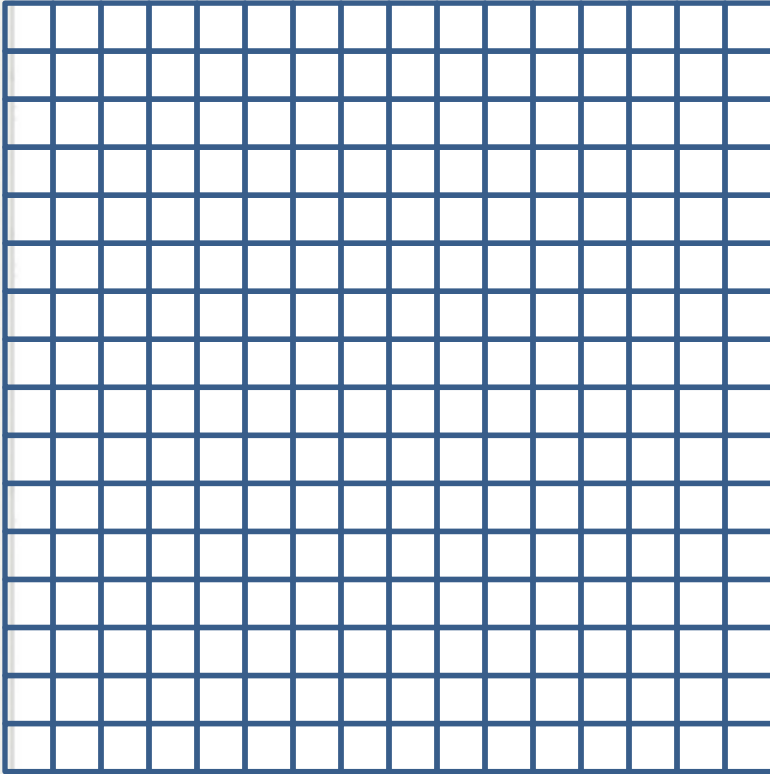
Blocking example



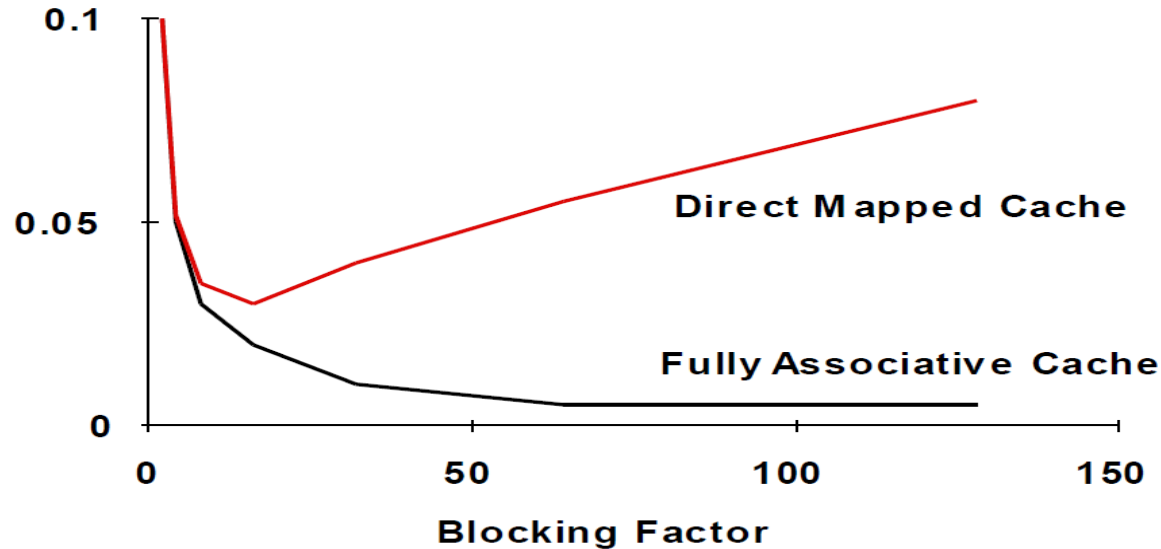
N transfers $\rightarrow N/B + N$ misses



Blocking example

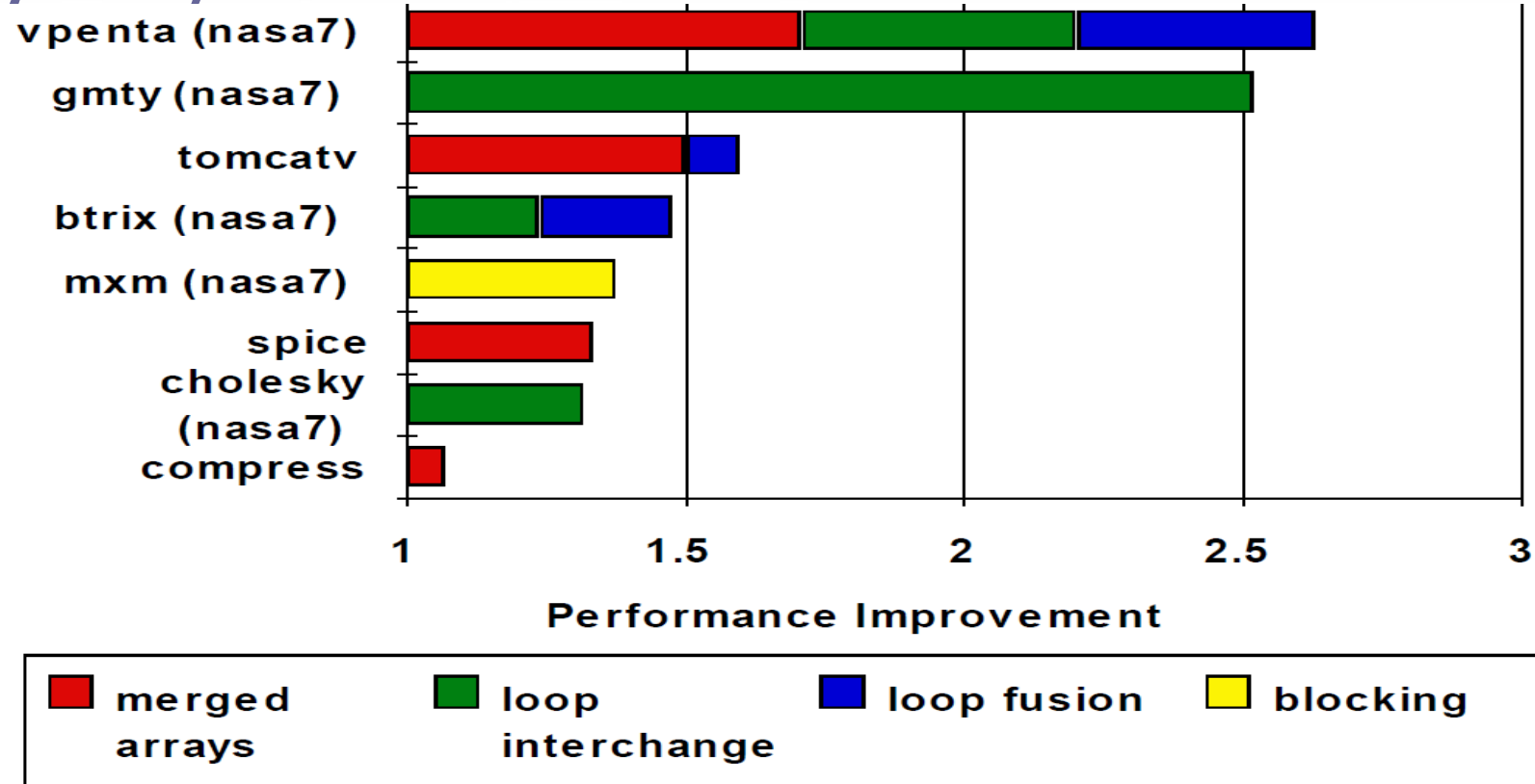


Reducing conflict misses by blocking



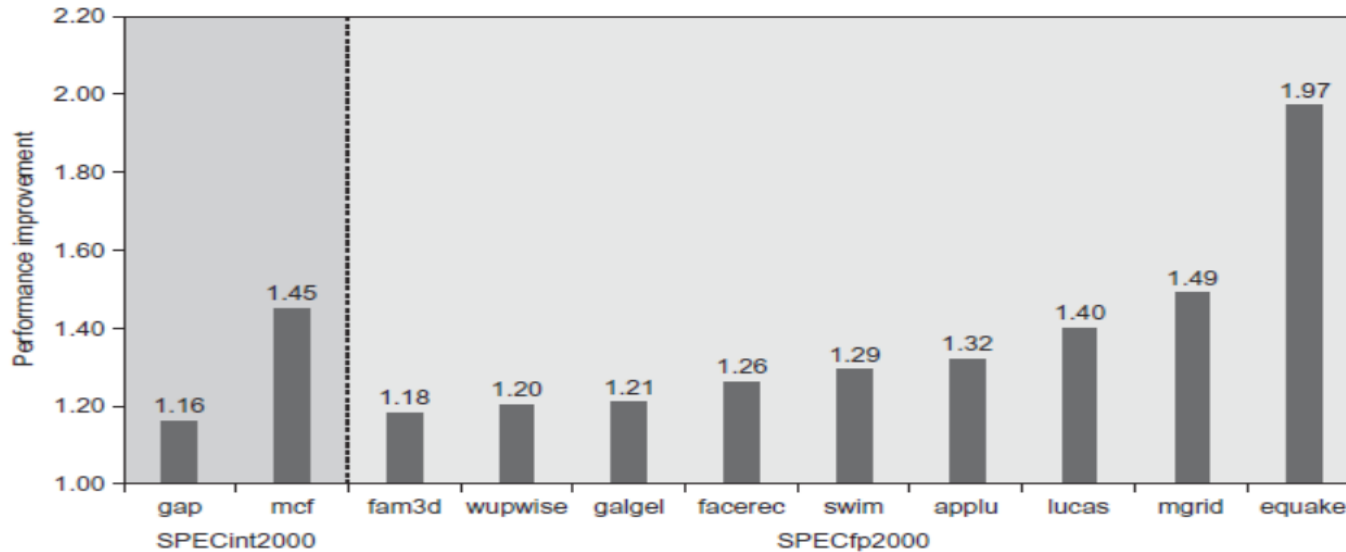
- Conflict misses in caches not FA vs. Blocking size
 - Lam et al [1991] a blocking factor of 24 had a fifth the misses vs. 48 despite both fit in cache

Summary of compiler optimizations to reduce cache misses (by hand)



Hardware Prefetching

- Fetch two blocks on miss (include next sequential block)



Pentium 4 Pre-fetching

Compiler prefetching

- Insert prefetch instructions before data is needed
- Non-faulting: prefetch doesn't cause exceptions
- Register prefetch
 - Loads data into register
- Cache prefetch
 - Loads data into cache
- Combine with loop unrolling and software pipelining

Reducing misses by software prefetching

- Data Prefetch
 - Load data into register (HP PA-RISC loads)
 - Cache Prefetch: load into cache (MIPS IV, PowerPC, SPARC v. 9)
 - Special prefetching instructions cannot cause faults; a form of speculative execution
- Issuing Prefetch Instructions takes time
 - Is cost of prefetch issues < savings in reduced misses?
 - Higher superscalar reduces difficulty of issue bandwidth

Reducing misses by software prefetching

```
for (i = 0; i < 3; i = i+1)
```

```
    for (j = 0; j < 100; j = j+1)
```

```
        a[i][j] = b[j][0] * b[j+1][0];
```

- Let's assume we have an 8 KB direct-mapped data cache with 16-byte blocks, and it is a write-back write allocate cache. The elements of a and b are 8 bytes long floating-point. There are 3 rows and 100 columns for a and 101 rows and 3 columns for b. They are not in the cache at the start of the program.

Reducing misses by software prefetching

- Elements of a are written in the order that they are stored in memory, so a will benefit from spatial locality: The even values of j will miss and the odd values will hit. Since a has 3 rows and 100 columns, its accesses will lead to $3 \times (100/2)$, or 150 misses.
- The array b does not benefit from spatial locality since the accesses are not in the order it is stored. The array b does benefit twice from temporal locality: The same elements are accessed for each iteration of i , and each iteration of j uses the same value of b as the last iteration. Ignoring potential conflict misses, the misses due to b will be for $b[j+1][0]$ accesses when $i = 0$, and also the first access to $b[j][0]$ when $j = 0$. Since j goes from 0 to 99 when $i = 0$, accesses to b lead to $100 + 1$, or 101 misses.
- Thus, this loop will miss the data cache approximately 150 times for a plus 101 times for b , or 251 misses.

Reducing misses by software prefetching

```
for (j = 0; j < 100; j = j+1){
    prefetch(b[j+7][0]);
    /* b(j,0)for 7 iterations later */
    prefetch(a[0][j+7]);
    /* a(0,j)for 7 iterations later */
    a[0][j] = b[j][0] * b[j+1][0];
};
for (i = 1; i < 3; i = i+1)
    for (j = 0; j < 100; j = j+1){
        prefetch(a[i][j+7]);
        /* a(i,j)for +7 iterations */
        a[i][j] = b[j][0] * b[j+1][0];
    }
```

Reducing misses by software prefetching

- This revised code prefetches $a[i][7]$ through $a[i][99]$ and $b[7][0]$ through $b[100][0]$, reducing the number of nonprefetched misses to
 - 7 misses for elements $b[0][0]$, $b[1][0]$, . . . , $b[6][0]$ in the first loop
 - 4 misses ($\lceil 7/2 \rceil$) for elements $a[0][0]$, $a[0][1]$, . . . , $a[0][6]$ in the first loop (spatial locality reduces misses to 1 per 16-byte cache block)
 - 4 misses ($\lceil 7/2 \rceil$) for elements $a[1][0]$, $a[1][1]$, . . . , $a[1][6]$ in the second loop
 - 4 misses ($\lceil 7/2 \rceil$) for elements $a[2][0]$, $a[2][1]$, . . . , $a[2][6]$ in the second loop
- or a total of 19 nonprefetched misses. The cost of avoiding 232 cache misses is executing 400 prefetch instructions, likely a good trade-off.

Use HBM to Extend Hierarchy



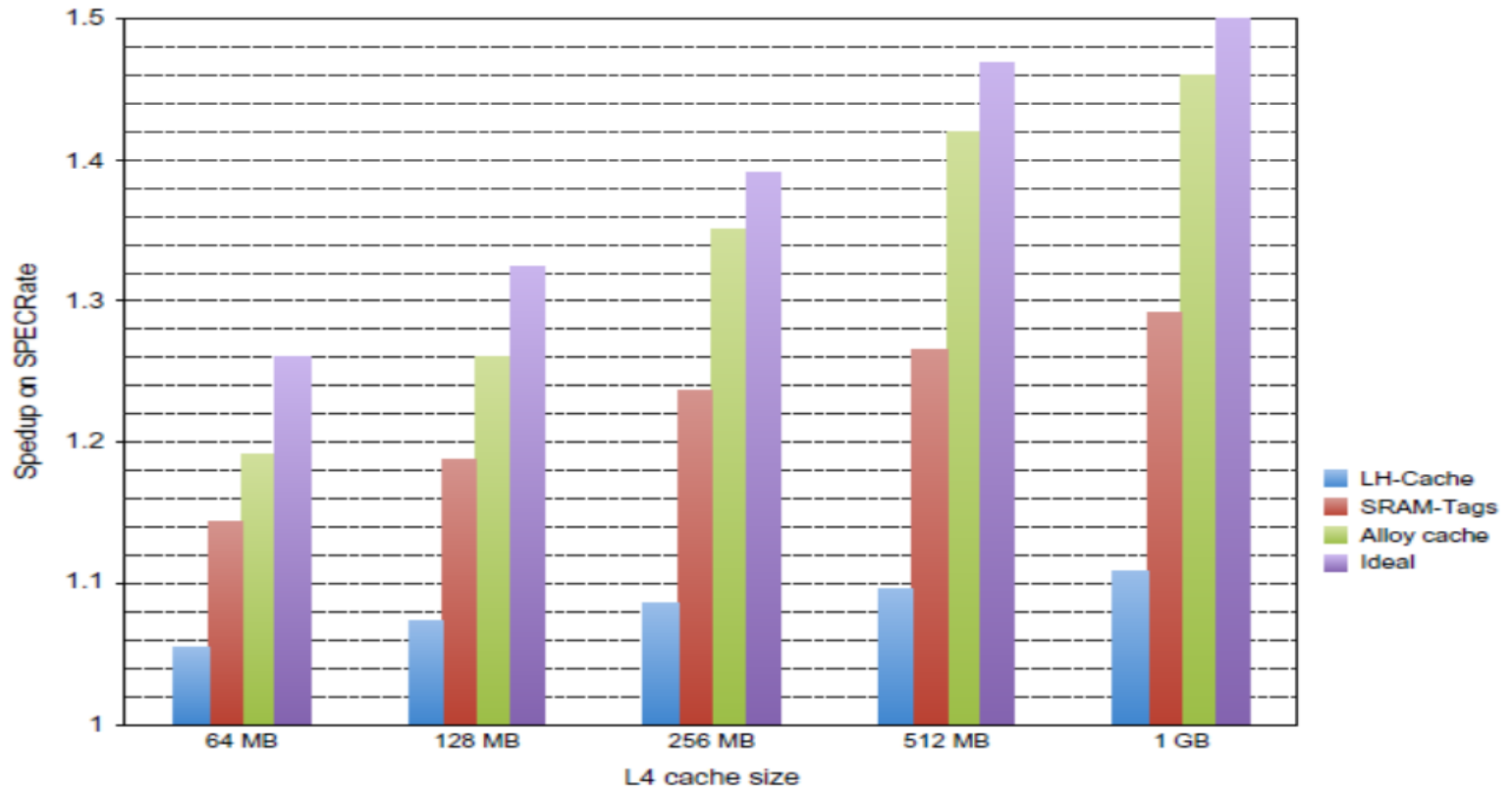
- 128 MiB to 1 GiB
- Smaller blocks require substantial tag storage
- Larger blocks are potentially inefficient
- One approach (*L-H*):
 - Each SDRAM row is a block index
 - Each row contains set of tags and 29 data segments
 - 29-set associative
 - Hit requires a CAS

High Bandwidth Memory - high-performance RAM interface for 3D-stacked DRAM

Use HBM to Extend Hierarchy

- Another approach (*Alloy cache*):
 - Mold tag and data together
 - Use direct mapped
- Both schemes require two DRAM accesses for misses
 - Two solutions:
 - Use map to keep track of blocks
 - Predict likely misses

Use HBM to Extend Hierarchy



Summary

Technique	Hit time	Band-width	Miss penalty	Miss rate	Power consumption	Hardware cost/complexity	Comment
Small and simple caches	+			—	+	0	Trivial; widely used
Way-predicting caches	+				+	1	Used in Pentium 4
Pipelined & banked caches	—	+				1	Widely used
Nonblocking caches		+	+			3	Widely used
Critical word first and early restart			+			2	Widely used
Merging write buffer			+			1	Widely used with write through
Compiler techniques to reduce cache misses				+		0	Software is a challenge, but many compilers handle common linear algebra calculations
Hardware prefetching of instructions and data			+	+	—	2 instr., 3 data	Most provide prefetch instructions; modern high-end processors also automatically prefetch in hardware
Compiler-controlled prefetching			+	+		3	Needs nonblocking cache; possible instruction overhead; in many CPUs
HBM as additional level of cache		+/-	—	+	+	3	Depends on new packaging technology. Effects depend heavily on hit rate improvements

Замена блокова кеш меморије

- При избору алгоритма треба водити рачуна:
 - Алгоритам треба да обезбеди **минималну вероватноћу** да ће блок који је одабран за замену и враћен из кеш у оперативну меморију убрзо морати поново да се довуче из оперативне у кеш меморију.
 - Други је да **цена хардвера** потребног за његову реализацију буде што је могуће нижа.
- Код кеш меморије са директним пресликавање се не корити.

Замена блокова кеш меморије

Разматрају две групе алгорита замене и то

- Приближни LRU алгоритама
 - Замена на основу једног бита 1b
 - MRU псеудо LRU алгоритама (NRU)
 - Модификовани псеудо LRU алгоритама – MPLRU
 - SIDE алгоритама
- Побољшани LRU алгоритама
 - Сегментирани LRU
 - 2Q
 - LRU-K
 - Adaptive Replacement Cache (ARC)

Замена на основу једног бита 1b

- Идеја: Сетови су подељени на две групе, а приликом избацивања се из оне групе којој није последње приступано на случајан начин бира један улаз. Циљ је да се заштити улаз коме је најскорије приступано, и његови суседи, од избацивања.
- Могућа реализација: користи се један бит по сету који памти којој да ли се приступало првој половини улаза или другој половини. За случајан избор се користи глобални FIFO бројач.
- Лоше стране: не води се довољно рачуна о томе да ће блок који је замењен можда убрзо морати поново да се довуче из оперативне у кеш меморију.
- Добре стране: хардвер за реализацију је **једноставан**.

MRU псеудо LRU алгоритам - NRU

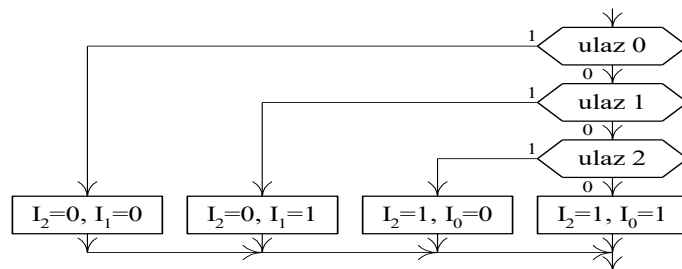
- Идеја: За сваки блок се памти да ли је најскорије приступани, све док се не приступи свим блоковима, када се сви индикатори (осим овог коме се последње приступа) ресетују.
- Могућа реализација: За сваки улаз се памти по један бит. Када се приступи неком улазу тај бит се поставља на вредност 1. Уколико су сви остали бити постављени на вредност 1 онда се постављају на вредност 0. За замену се бира улаз са најмањим редним бројем у коме је вредност 0.
- Лоше стране: памти ограничену историју.
- Добре стране: хардвер за реализацију је **једноставан**. (Колико?)

Модификовани псеудо LRU алгоритам

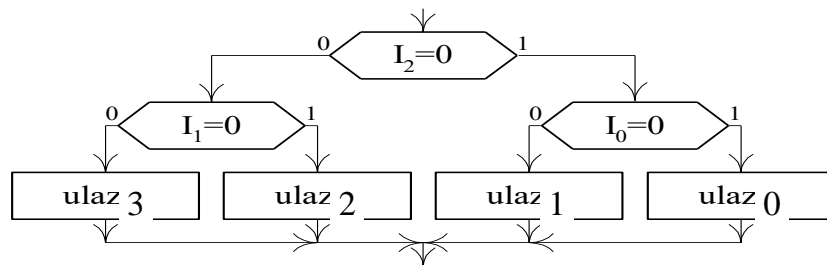
- Идеја: Како би се сачувала што дужа историја користити псеудо LRU алгоритам али са претходно одабраном групом.
- Могућа реализација: Алгоритам замене често се користи када је број улаза по сету најчешће четири или осам. У случају четири улаза по сету потребан регистар дужине четири бита и пратећа комбинациона логика (један бит садашњег приступа, један бит за претходни приступ, и по једна бит за сваку групу)



Псеудо LRU алгоритам - подсетник

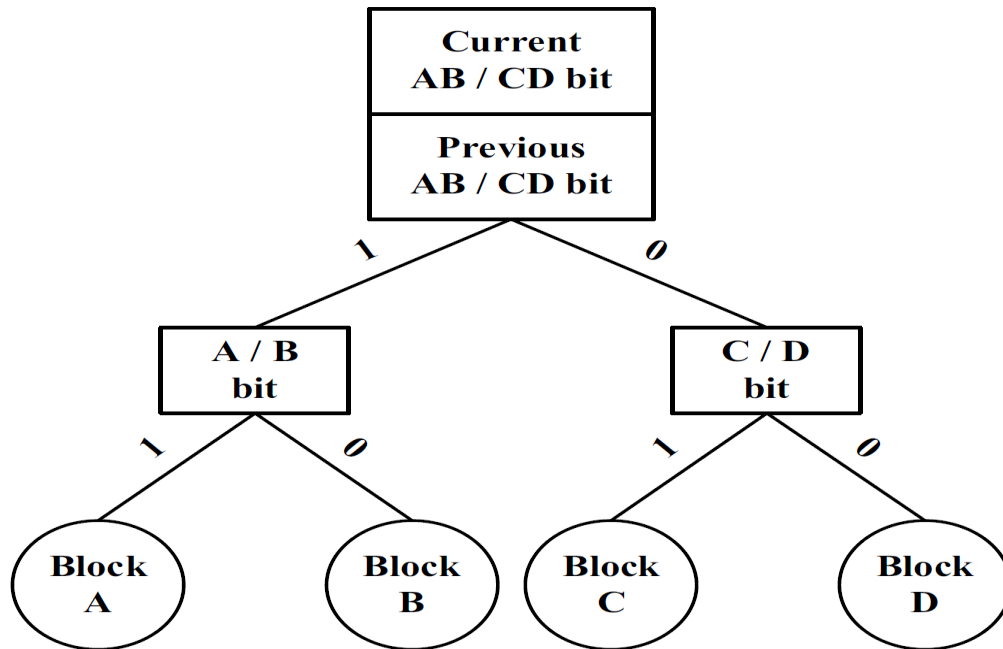


Дијаграм тока при ажурирању



Дијаграм тока при одређивању улаза за замену

Модификовани псеудо LRU алгоритам



Дијаграм тока при одређивању улаза за замену

Модификовани псеудо LRU алгоритам

- Добре стране: Овај алгоритам је заснован на истој претпоставци као и LRU и псеудо LRU алгоритам и има за циљ да за замену бира блок коме се најмање скоро приступало. Има нешто боље карактеристике у односу на псеудо LRU алгоритам.
- Лоше стране: Хардвер за његову реализацију је једноставнији у односу на хардвер LRU алгоритма, па је овај алгоритам **мање прецизан** у односу на LRU алгоритам.

SIDE алгоритам

- Идеја: Коришћење бројачког регистра (као FIFO) који се ажурира током приступа (слично LRU). На овај начин бројачки регистар дели улазе на две групе: могући најскорије приступани са леве стране бројача и најдавніје приступани са десне стране бројача.
- Уколико има сагласност у улазу i ,
 - онда уколико је $i \geq c$ бројач се поставља на $(i+1) \bmod N$,
 - иначе се на мења.
- Уколико се не открије сагласност
 - блок се довлачи у неки улаз са десне стране бројача (специјалан случај довлачи се у улаз c)
 - бројач се поставља на $(c+1) \bmod N$.

SIDE алгоритам

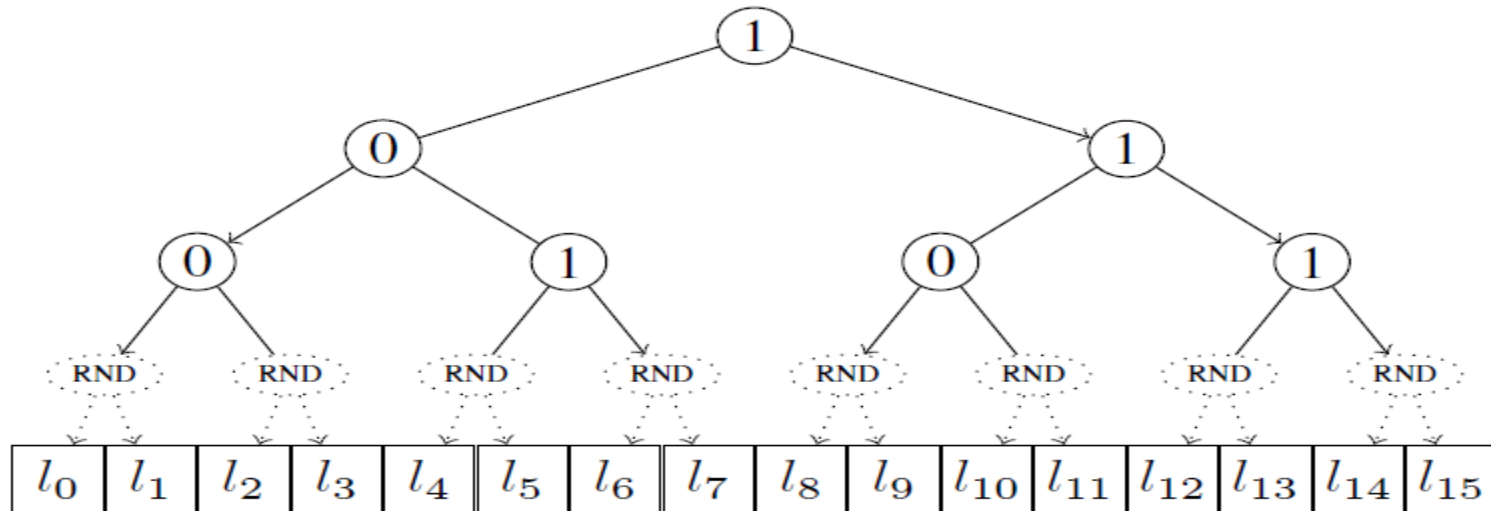
- Добре стране: Комбинација FIFO алгоритма у погледу хардвера и LRU алгоритма у погледу замене. Хардвер за реализацију је једноставнији у односу на хардвер LRU алгоритма
- Лоше стране: На почетку сваке фазе претходне информације се губе. Приликом избора улаза за замену бира се **неки** улаз са десне стране бројача. Алгоритам је **мање прецизан** у односу на LRU алгоритам.

Преглед приближних LRU алгоритама

Policy	Bits required	4-way	What to do when hit on i?	What to do when miss?	Logic required (only estimations)	Performance to LRU
Random	$\log_2(N_{ways})$	2	-	Update LFSR	LFSRs and XORs to generate the random	22% worse [ZOU04]
FIFO	$N_{sets} \cdot \log_2(N_{ways})$	2. N_{sets}	-	Increment FIFO counter	Incrementing logic and $N_{sets} \log_2(N_{ways})$ -bit-counter registers. Low overhead in principle.	12-20% worse [ZOU04, SMI83]
LRU	$N_{sets} \cdot N_{ways} \cdot \log_2(N_{ways})$	8. N_{sets}	Update the LRU stack	Update the LRU stack	Can be implemented in a matrix form. Seems too intricate and computation time consuming for embedded systems	-
1-bit	N_{sets}	1. N_{sets}	Update the bit	Update the bit	LFSRs and XORs to generate the random in the LRU partition, $N_{sets} \cdot N_{ways}$ OR/AND	10-20% worse for high associativity; 5-10% worse for low [SO88]
MRU (NRU)	$N_{sets} \cdot N_{ways}$	4. N_{sets}	Update the MRU bits	Update the MRU bits	Around $N_{ways} \cdot N_{sets}$ registers (for the MRU bits), $N_{ways} \cdot N_{sets}$ muxes, $3 \cdot N_{sets} \cdot \log_2(N_{ways})$ AND/OR	1% worse to 3% better [ZOU04]
PLRUt	$N_{sets} \cdot (N_{ways} - 1)$	3. N_{sets}	Update the tree bits	Update the tree bits	Around $N_{sets} \cdot (N_{ways} - 1)$ registers, $N_{sets} \cdot (N_{ways} - 1)$ Muxes, $N_{sets} \cdot (N_{ways} - 1)$ inverters, $N_{sets} \cdot \log_2(N_{ways})$ OR	1-5% worse [ZOU04]
MPLRU	$N_{sets} \cdot N_{ways}$	4. N_{sets}	Update the tree bits	Update the tree bits		
SIDE	$N_{sets} \cdot \log_2(N_{ways})$	2. N_{sets}	Update the counter c to $(i+1) \bmod S$ if $i \geq c$ Nothing if $i < c$	Increment counter and random selection	$N_{sets} \log_2(N_{ways})$ -bit-registers for the counters. N_{ways} -bit comparator and incrementation. LFSRs and XORs to generate the random during the discarding selection.	0-20% worse. Strongly dependent on associativity. On low associativity (2-8), 0-5% [DEV90]

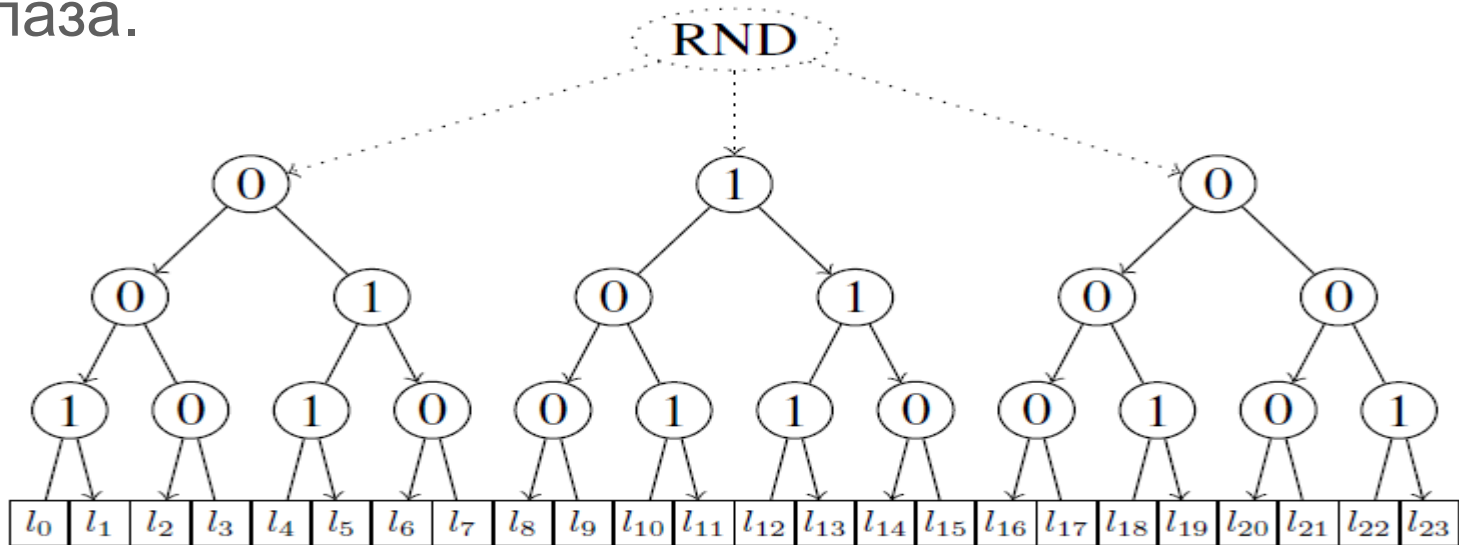
Псеудо LRU random алгоритам

- Идеја: Користити псеудо LRU алгоритам како би се дошло до групе могућих улаза. Унутар групе на случајан начин изабрати један улаз.



Random псеудо LRU алгоритам

- Идеја: Користећи случајан избор одабрати једну од могућ група улаза (*Ordered selection*). Унутар групе користећи псеудо LRU алгоритам изабрати један улаза.

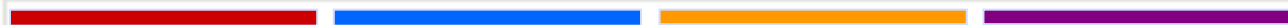


LFU алгоритам (*Least frequently used*)

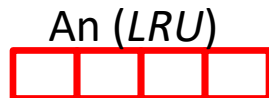
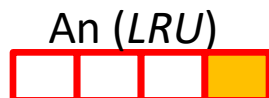
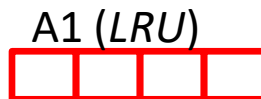
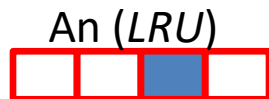
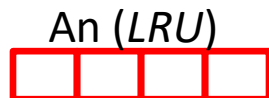
- Идеја: За замену се бира блок коме се најмањи број пута приступало.
- Могућа реализација: Додељивање бројача сваком блоку који се учитава у кеш меморију. Сваки пут када се приступи блоку, бројач се повећава за један. Када се кеш меморија напуни и појави се захтев за новим блоком тражи се блок са најмањом вредношћу бројача и тај блок се уклања из кеша.
- Проблеми: Посматра се блок у меморији коме се пуно пута приступило у кратком временском периоду и коме се више не приступа. Због брзине којом се приступило, бројач се драстично повећао иако се неће поново користити. Ово оставља друге блокове који се заправо могу чешће користити подложним избацивању. Ово значи да се нови блокови који су скоро учитани у кеш подложне брзом уклањању, јер почињу са ниским бројачем, иако би се након тога могле врло често користити.

SLRU алгоритам (*Segmented LRU*)

- Идеја: Кеш се дели на два сегмента фиксне величине
 - заштићени сегменти и
 - могући сегмент.
- У случају да нема сагласности податак се додаје у део могућег сегмента који је најскорије коришћен (*MRU*).
- У случају да има сагласности податак се ставља у део заштићеног сегмента који је најскорије коришћен (*MRU*).
- Пошто заштићени сегмент има коначну величину додавање нових блокова у заштићени сегмент постојеће блокове према LRU се пребацују у најскорије коришћен део могућег сегмента.



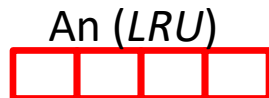
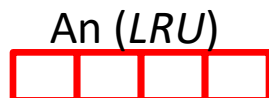
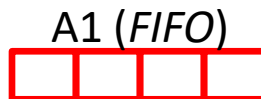
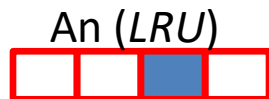
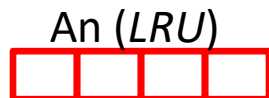
SLRU алгоритм (*Segmented LRU*)



Поједностављени 2Q алгоритам

- Идеја: Кеш се дели на два реда променљиве величине, код којих први садржи само блокове којима је приступано једном $A1$ ($FIFO$) и други који садржи блокове којима је приступано више пута A_n (LRU).
 - Уколико има сагласности:
 - Ако је у A_n блок ставити га на почетак A_n реда.
 - Ако је у $A1$ блок избацити га из $A1$ и пребацити га на почетак A_n .
 - Уколико нема сагласности:
 - Ако нема места и величина $A1$ је изнад прага избацити блок из $A1$.
 - Ако нема места и величина $A1$ је испод прага избацити блок из A_n .
- У сваком случају блок ставити на почетак $A1$ реда.

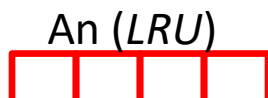
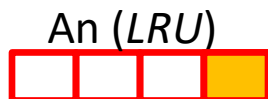
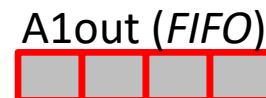
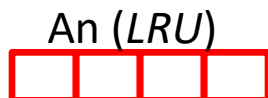
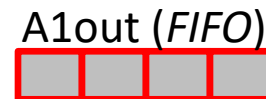
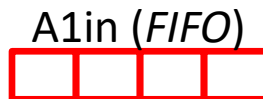
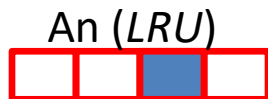
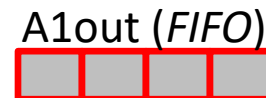
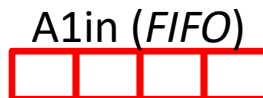
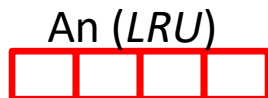
Поједностављени 2Q алгоритам



2Q алгоритам – фиксне величине

- Идеја: Коришћење два реда код којих први садржи само блокове којима је приступано једном $A1$ ($FIFO$) и други који садржи блокове којима је приступано више пута A_n (LRU). Ред $A1$ је подељен у два $FIFO$ реда $A1in$ (блокови) и $A1out$ (само референце).
- Уколико има сагласности:
 - Ако је у A_n ажурирати LRU .
 - Ако је у $A1out$ пребацити га у A_n , блок која се избацује из A_n иде у $A1in$, а референца на блок који се избацује из $A1in$ иде у $A1out$, ажурирати LRU и оба $FIFO$.
 - Ако је у $A1in$ не радити ништа.
- Уколико нема сагласности:
 - Блок се довлачи у $A1in$, референца на блок који се избацује из $A1in$ иде у $A1out$, ажурирати оба $FIFO$.

2Q алгоритм – фиксне величине



2Q алгоритам

- Идеја: Коришћење два реда код којих први садржи само блокове којима је приступано једном $A1$ ($FIFO$) и други који садржи блокове којима је приступано више пута A_n (LRU). Ред $A1$ је подељен у два $FIFO$ реда $A1in$ (блокови) и $A1out$ (само референце).
- Уколико има сагласности:
 - Ако је у A_n тај блок ставити на почетак A_n реда.
 - Ако је у $A1out$, ослободити место, блок ставити на почетак A_n реда.
 - Ако је у $A1in$ не радити ништа.
- Уколико нема сагласности:
 - Ослободити место, ставити на почетак $A1in$.
- Ослободити место:
 - Ако нема места и ако је величина $A1in$ изнад прага избацити блок из $A1in$ и његову ставити референцу у $A1out$, а ако је величине $A1out$ изнад прага избацити стару референцу из $A1out$.
 - Ако нема места иначе избацити блок из A_n .

LRU-K

- Идеја: Подела кеш меморије на K делова који одговарају блоковима којима је приступано између 1 и K пута недавно.
- Приликом сваког приступа потребно је водити евиденцију о броју приступа у интервалу (логика за премештање). У случају да нема сагласности за избацивање се бита онај улаз коме је најмање приступано у интервалу и који међу њима има најмању LRU.

ARC (*Adaptive Replacement Cache*)

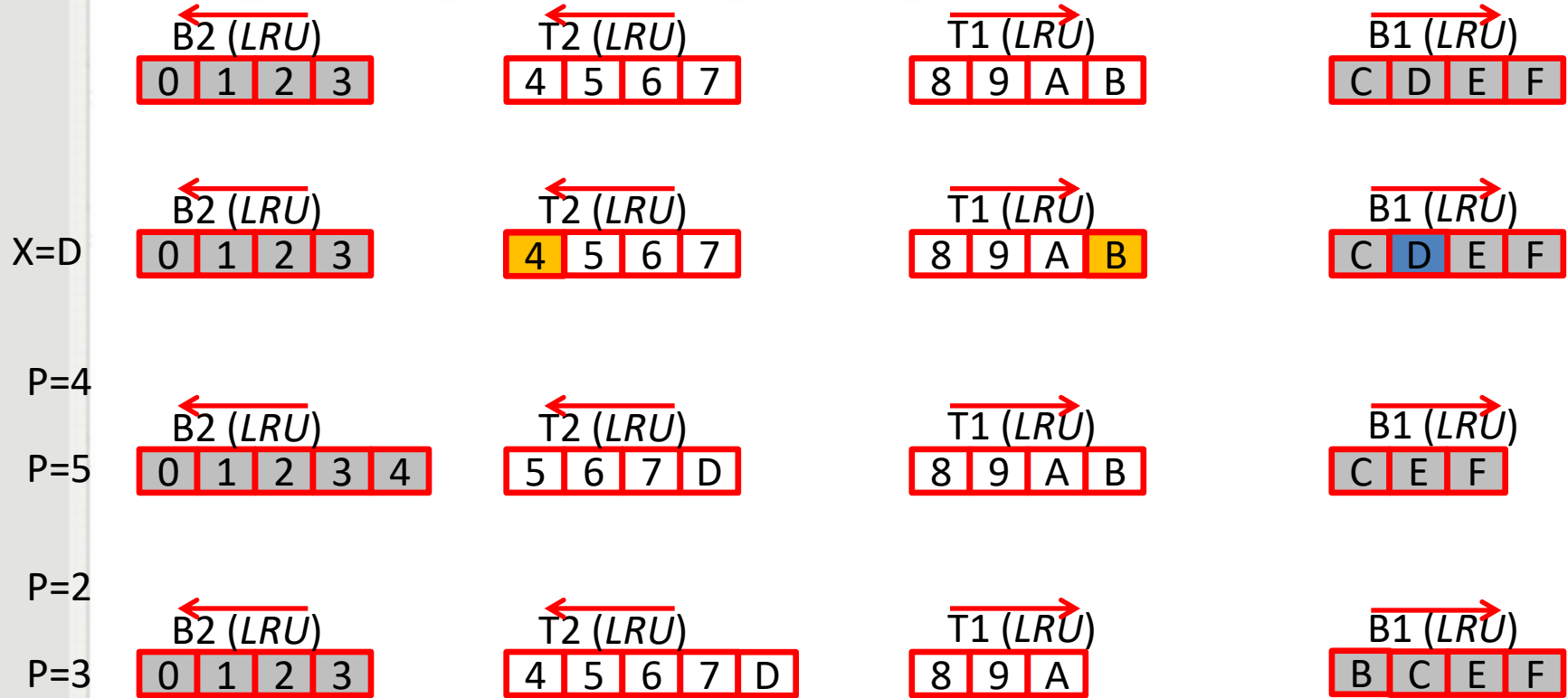
- Идеја: Кеш се дели на две групе: L1 и L2 свака дужине c . L1 садржи блокове којима је скоро приступано једном и L2 садржи блокове којима је скоро приступано барем два пута. Ове две групе се онда динамички деле у по две подгрупе T(op)– која садржи MRU део – и B(ottom)– који садржи LRU део – тако да важи $||T1 \cup T2||=c$.
- T1 и T2 чувају блокове, док B1 и B2 чувају референце скоро избачених блокова.
- За величину подгрупе T1 се користи параметар p . У било ком тренутку ARC алгоритам се понаша као алгоритам са фиксном величином при чему је p блокова у T1 и $c-p$ у T2. Параметар p се динамички мења како би се чувала најактивнија листа.

ARC (*Adaptive Replacement Cache*)

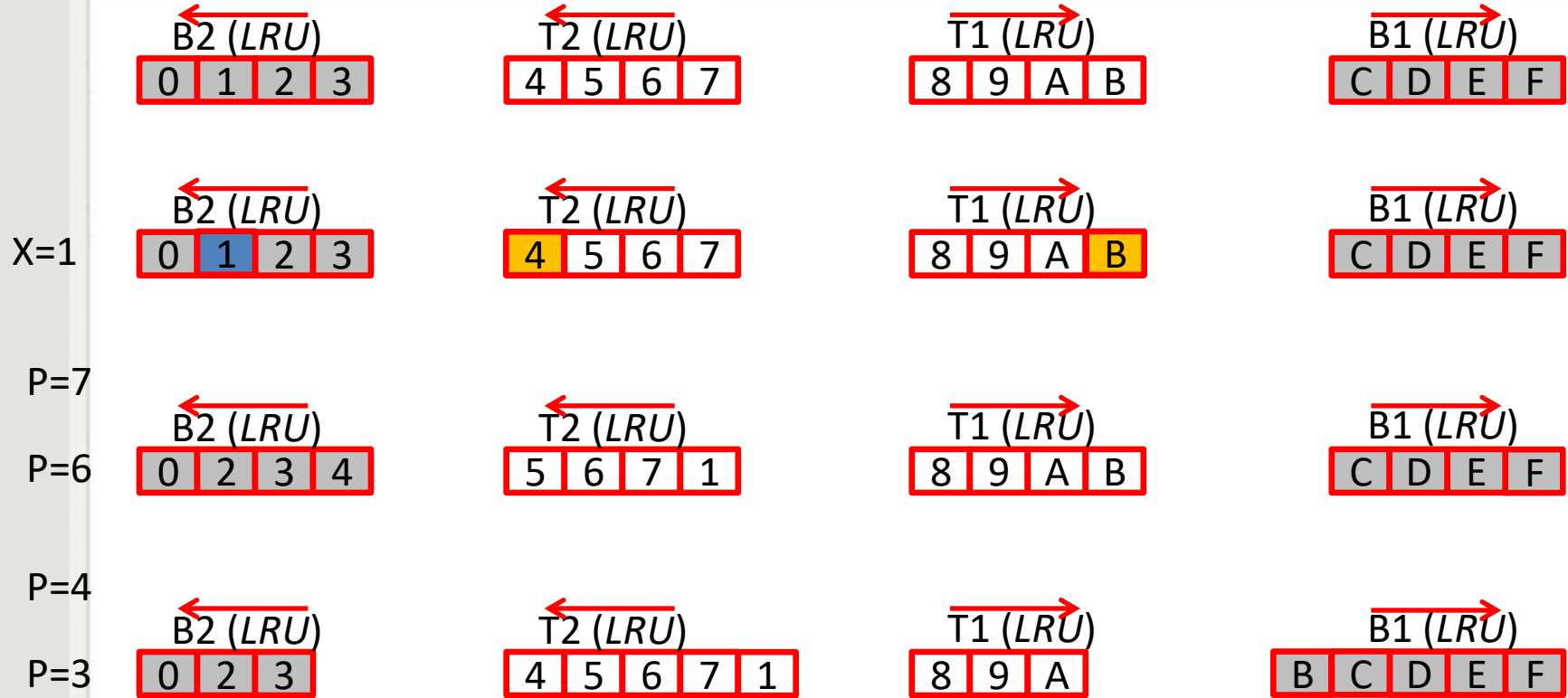
$p = 0$.

- Уколико има сагласности:
 - Ако је у $T1$ или $T2$, пребаци на почетак $T2$
 - Ако је у $B1$, $p = \min(c, p + \max(|B2|/|B1|, 1))$, ослободи простор, пребаци на почетак $T2$
 - Ако је у $B2$, $p = \max(0, p - \max(|B1|/|B2|, 1))$, ослободи простор, пребаци на почетак $T2$
 - Уколико нема сагласности:
 - Ако је $|L1|=c$
 - Ако је $|T1|<c$, избаци из $B1$, ослободи простор
 - Ако је $|T1|=c$, избаци из $T1$
 - Ако је $|L1|<c$ and $|L1|+|L2| \geq c$
 - Ако је $|L1|+|L2| = c$, избаци из $B2$, ослободи простор
- у сваком случају пребаци на почетак $T1$
- Ослобађање простора:
 - Ако је $|T1|>1$ and ((сагласност у $B2$ and $|T1|=p$) or ($|T1|>p$)) избаци из $T1$ у $B1$
 - иначе избаци из $T2$ у $B2$

ARC (Adaptive Replacement Cache)



ARC (Adaptive Replacement Cache)



Преглед побољшаних LRU алгоритама

Policy	Lists	User defined parameters	What to do when hit on i?	What to do when miss?	Complexity per request	Performance to LRU
SLRU	2 LRU lists of fixed size $c_1+c_2=c$	PbS size	Move i to the MRU position of the PtSd. Move the LRU line of the PtS to the MRU position of the PbSe if necessary. Discard the LRU line of the PbS if necessary.	Move the line to the MRU part of the PbS. Discard the LRU line of the PbS if necessary.	Constant	Around 5% better [KAR94]
2Q	1 LRU list and 1 FIFO list split into two sub lists	Kin and Kout	Move it the MRU side of the LRU list Am. If it was in A1, remove it from A1.	Move the line to the back-end of the FIFO A1in. Move the discarded line to the FIFO A1out. Discard line from this FIFO.	Constant	5-10% better [JOH94]
LRU-K	K history lists	CIP	Update the CRP of the data and its K history lists.	Select a victim by exploring the list of 1st and K access. Initialize the CRP and the K history lists of the data.	Logarithmic time (because of its priority queue)	Around 50% better for very large database buffers [ONE93]
ARC	2 LRU lists of size c	-	Move i to the MRU position of T2	Hit on B1/B2: Update p (addition and min/max operations) and move i to the MRU position of T2. Miss on T1/B1/T2/B2: Delete the LRU page in B1 if $ T1 < c$, in T1 otherwise. Move i to the MRU in T1.	Constant	50-200% better [MEG03]

<https://people.kth.se/~ingo/MasterThesis/ThesisDamienGille2007.pdf>

Питања?

Електротехнички Факултет
Универзитет у Београду

