

SEMINAR

Delhi Technological University
(COE-416)

Energy-Efficient Hardware Data Prefetching

Made by- Himanshu Koli (2K10/CO/041)
Hiren Madan (2K10/CO/042)

Contents

- ❑ Introduction
- ❑ What is Data Prefetching?
- ❑ Prefetching Classification
- ❑ How Prefetching works?
- ❑ Software Prefetching
- ❑ Limitations of Software based Prefetching
- ❑ Hardware Prefetching
- ❑ Hardware Vs. Software Approach
- ❑ Energy Aware Data Prefetching
- ❑ Energy Aware Prefetching Architecture
- ❑ Energy Aware Prefetching Techniques
- ❑ References

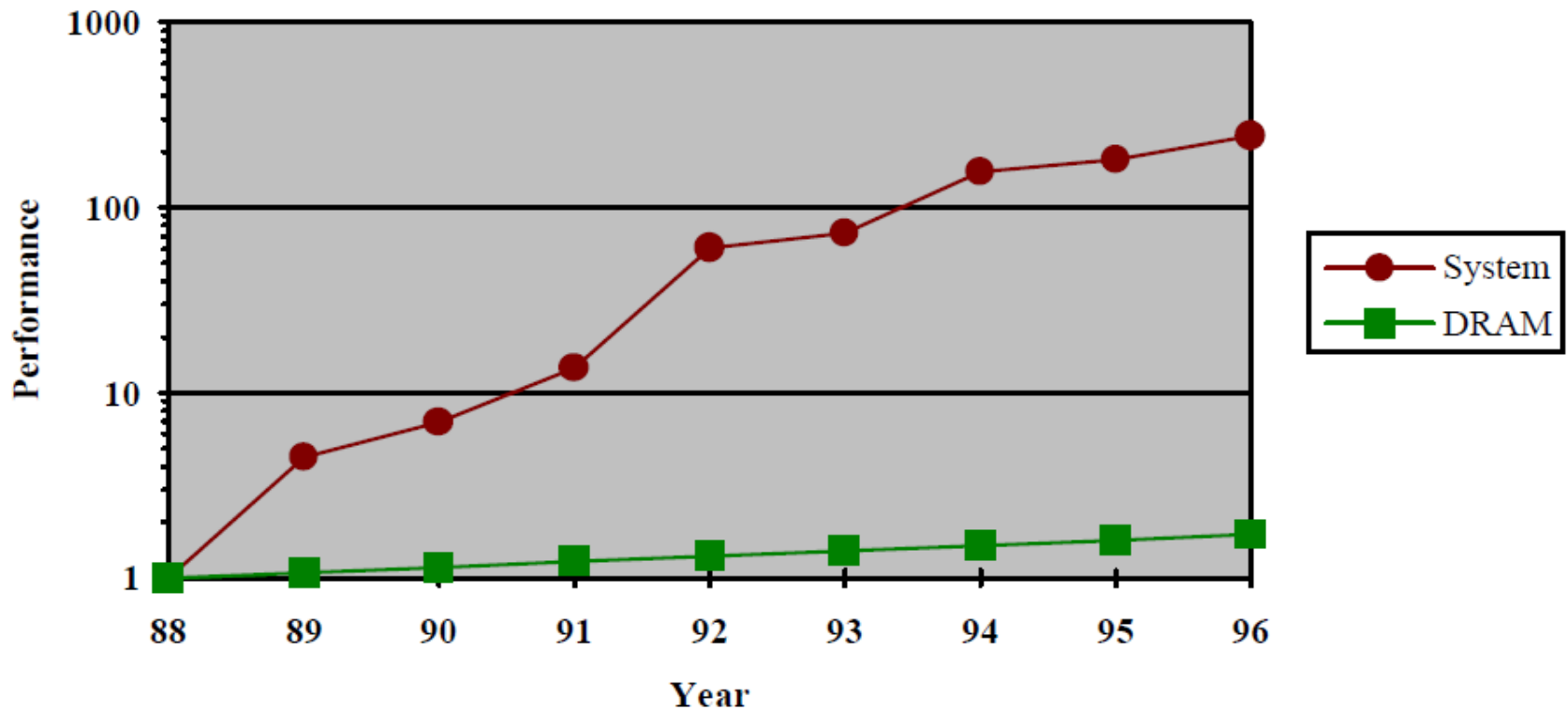
Introduction

Why need Data Prefetching?

- ❑ Microprocessor performance has increased at a dramatic rate .
- ❑ Expanding gap between microprocessor and DRAM performance has necessitated the use of aggressive techniques designed to reduce the large latency of memory accesses.
- ❑ Use of cache memory hierarchies have managed to keep pace with processor memory request rates but continue to be too expensive for a main store technology.
- ❑ Use of large cache hierarchies has proven to be effective in reducing the average memory access penalty for programs that show a high degree of locality in their addressing patterns .

But scientific and other data-intensive programs spend more than half their run times stalled on memory requests.

Processor-Memory Performance Gap

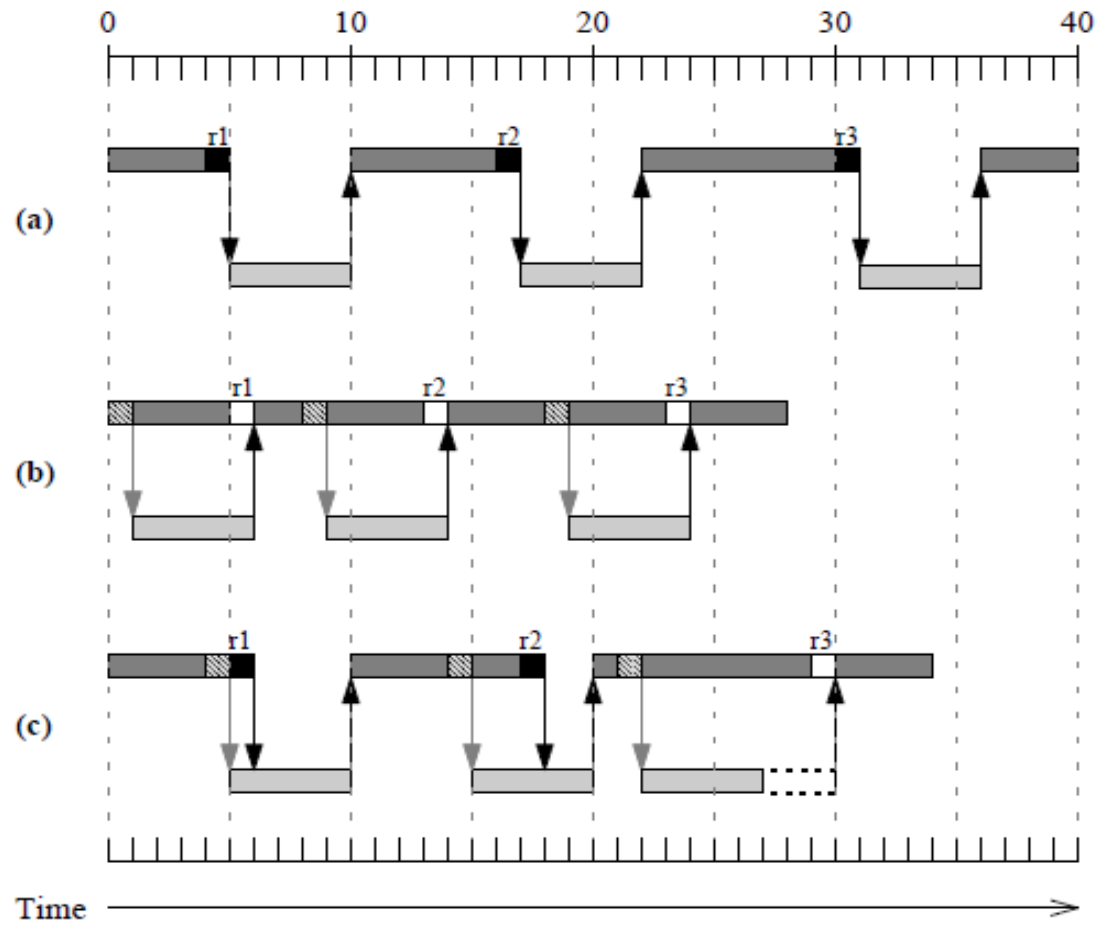


- ❑ On demand fetch policy, it will always result in a cache miss for the first access to a cache block. Such cache misses are known as *cold start or compulsory misses*.
- ❑ When we reference a large array, there is a high possibility of the elements of the array to be overwritten.
- ❑ If we need the previous value of the array which has been overwritten, then the processor needs to make full main memory access. This is called as *capacity miss*.

What is Data Prefetching ?

- ❑ *Data prefetching* anticipates such misses and issues a fetch to the memory system in advance of the actual memory reference, rather than waiting for a cache miss to perform a memory fetch .
- ❑ Prefetch proceeds in parallel with processor computation, allowing the memory system time to transfer the desired data from main memory to the cache.
- ❑ Prefetch will complete just in time for the processor to access the needed data in the cache without stalling the processor.

Execution Diagram assuming- a) No Prefetching, b) Perfect Prefetching and c) Degraded Prefetching



How Prefetching Works?

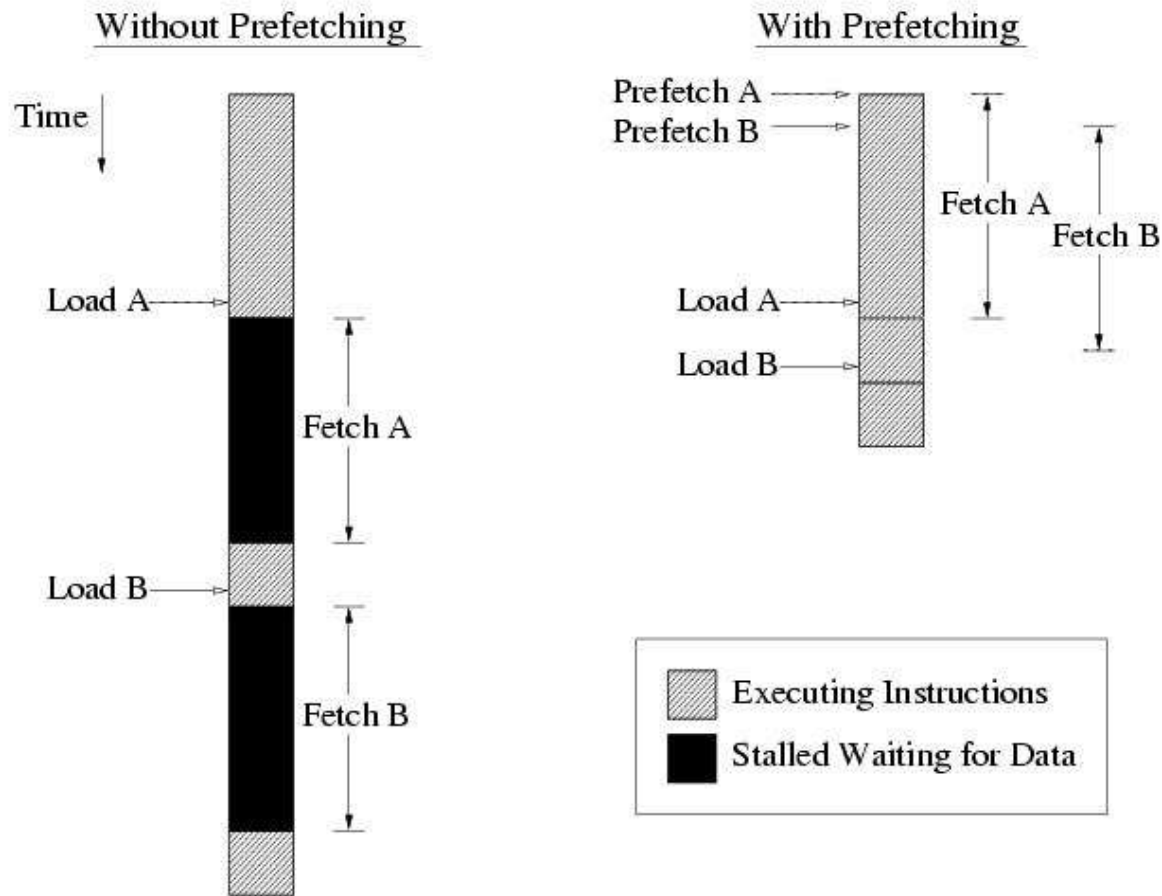



Figure 1.4: Illustration of how prefetching tolerates memory latency.

Prefetching Classification

- ❑ Various prefetching techniques have been proposed-
 - ❑ Instruction Prefetching vs. Data Prefetching
 - ❑ Software-controlled prefetching vs. Hardware-controlled prefetching.
- ❑ Data prefetching for different structures in general purpose programs:
 - ❑ Prefetching for array structures.
 - ❑ Prefetching for pointer and linked data structures.

Software Data Prefetching

- ❑ Explicit “fetch” instructions
 - ❑ Non-blocking memory operation.
 - ❑ Cannot cause exceptions (e.g. page faults).
- ❑ Additional instructions executed.
- ❑ Modest hardware complexity
- ❑ Challenge -- prefetch scheduling
 - ❑ Placement of *fetch* instruction relative to the matching load and store instruction.
 - ❑ Hand-coded by programmer or automated by compiler.

- 
- ❑ Adding just a few prefetch directives to a program can substantially improve performance.
 - ❑ Prefetching is most often used within loops responsible for large array calculations.
 - ❑ Common in scientific codes
 - ❑ Poor cache utilization
 - ❑ Predictable array referencing patterns
 - ❑ Fetch instructions can be placed inside loop bodies so that current iteration prefetches data for a future iteration.

Example : Vector Product

❑ No prefetching

```
for (i = 0; i < N; i++)  
{ sum += a[i]*b[i]; }
```

❑ Assume each cache block holds 4 elements .

❑ Code segment will cause a cache miss every fourth iteration.

❑ Simple prefetching

```
for (i = 0; i < N; i++)  
{  
    fetch (&a[i+1]);  
    fetch (&b[i+1]);  
    sum += a[i]*b[i];  
}
```

❑ Problem-

❑ Unnecessary prefetch operations

Example (contd.)

❑ Prefetching + loop unrolling

```
for (i = 0; i < N; i+=4)
{
    fetch (&a[i+4]);
    fetch (&b[i+4]);
    sum += a[i]*b[i];
    sum += a[i+1]*b[i+1];
    sum += a[i+2]*b[i+2];
    sum += a[i+3]*b[i+3];
}
```

❑ Problem

❑ First and last iterations

```
fetch (&sum);
fetch (&a[0]);
fetch (&b[0]);
for (i = 0; i < N-4; i+=4)
{
    fetch (&a[i+4]);
    fetch (&b[i+4]);
    sum += a[i]*b[i];
    sum += a[i+1]*b[i+1];
    sum += a[i+2]*b[i+2];
    sum += a[i+3]*b[i+3];
}
for (i = N-4; i < N; i++)
    sum = sum + a[i]*b[i];
```

Example (contd.)

- ❑ Previous assumption: prefetching 1 iteration ahead is sufficient to hide the memory latency.
- ❑ When loops contain small computational bodies, it may be necessary to initiate prefetches d iterations before the data is referenced.

$$d = \left\lceil \frac{l}{s} \right\rceil$$

d : prefetch distance

l : avg. memory latency

s : is the estimated cycle time of the shortest possible execution path through one loop iteration

```
fetch (&sum);
for (i = 0; i < 12; i += 4)
{
    fetch (&a[i]);
    fetch (&b[i]);
}

for (i = 0; i < N-12; i += 4)
{
    fetch(&a[i+12]);
    fetch(&b[i+12]);
    sum = sum + a[i] * b[i];
    sum = sum + a[i+1]*b[i+1];
    sum = sum + a[i+2]*b[i+2];
    sum = sum + a[i+3]*b[i+3];
}

for (i = N-12; i < N; i++)
    sum = sum + a[i]*b[i];
```

Limitation of Software-based Prefetching


- ❑ Normally restricted to loops with array accesses
- ❑ Hard for general applications with irregular access patterns
- ❑ Processor execution overhead
- ❑ Significant code expansion
- ❑ Performed statically.

Hardware Data Prefetching

- ❑ Special Hardware required.
- ❑ No need for programmer or compiler intervention.
- ❑ No changes to existing executable.
- ❑ Take advantage of run-time information.

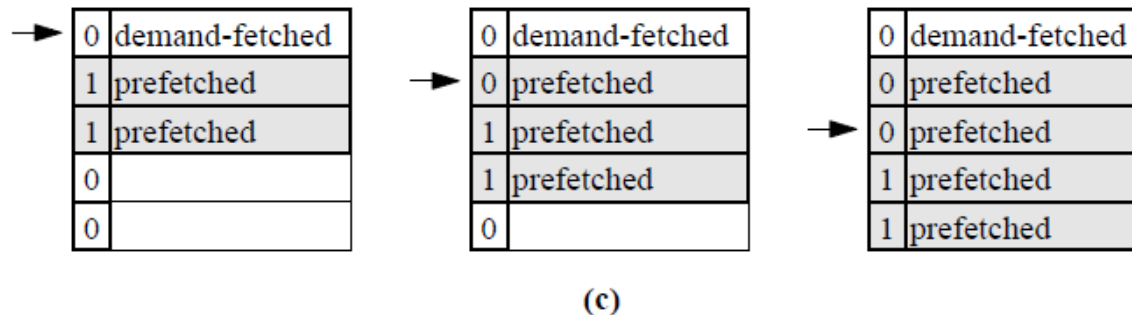
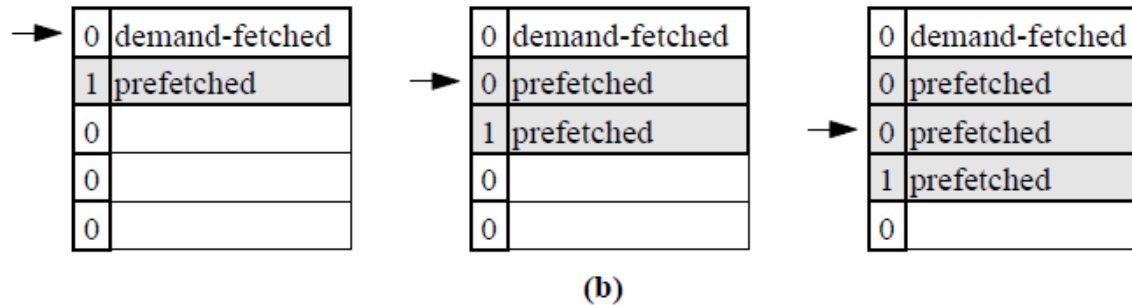
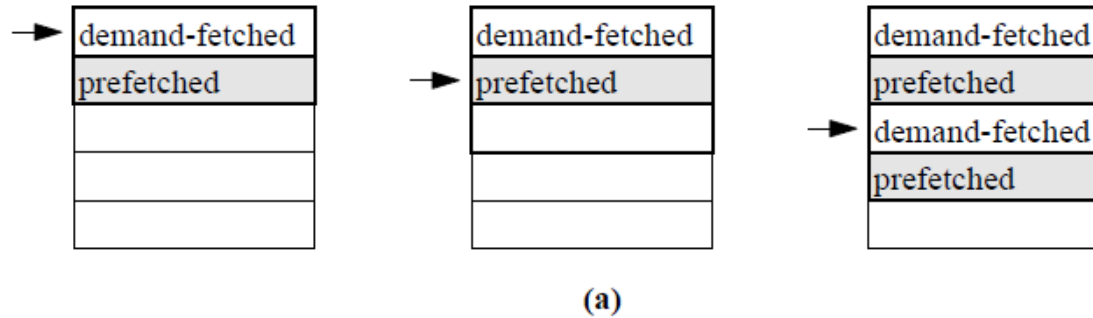
Sequential Prefetching

- ❑ By grouping consecutive memory words into single units, caches exploit the principle of spatial locality to implicitly prefetch data that is likely to be referenced in the near future.
- ❑ Larger cache blocks suffer from
 - ❑ cache pollution
 - ❑ false sharing in multiprocessors.
- ❑ One block *lookahead* (OBL) approach
 - ❑ Initiate a prefetch for block $b+1$ when block b is accessed.
 - ❑ Prefetch-on-miss
 - ❑ Whenever an access for block b results in a cache miss
 - ❑ Tagged prefetch
 - ❑ Associates a tag bit with every memory block.
 - ❑ When a block is demand-fetched or a prefetched block is referenced for the first time, next block is fetched.
 - ❑ Used in HP PA7200

- 
- OBL may not be initiated far enough in advance of the actual use to avoid a processor memory stall.
 - To solve this, increase the number of blocks prefetched after a demand fetch from one to K , where K is known as the *degree of prefetching*.
 - Aids the memory system in staying ahead of rapid processor requests.
 - As each prefetched block, b , is accessed for the first time, the cache is interrogated to check if blocks $b+1, \dots, b+K$ are present in the cache and, if not, the missing blocks are fetched from memory.

Three Forms of Sequential Prefetching:

a) Prefetch on miss, b) Tagged Prefetch and
c) Sequential Prefetching with $K = 2$.



❑ Shortcoming

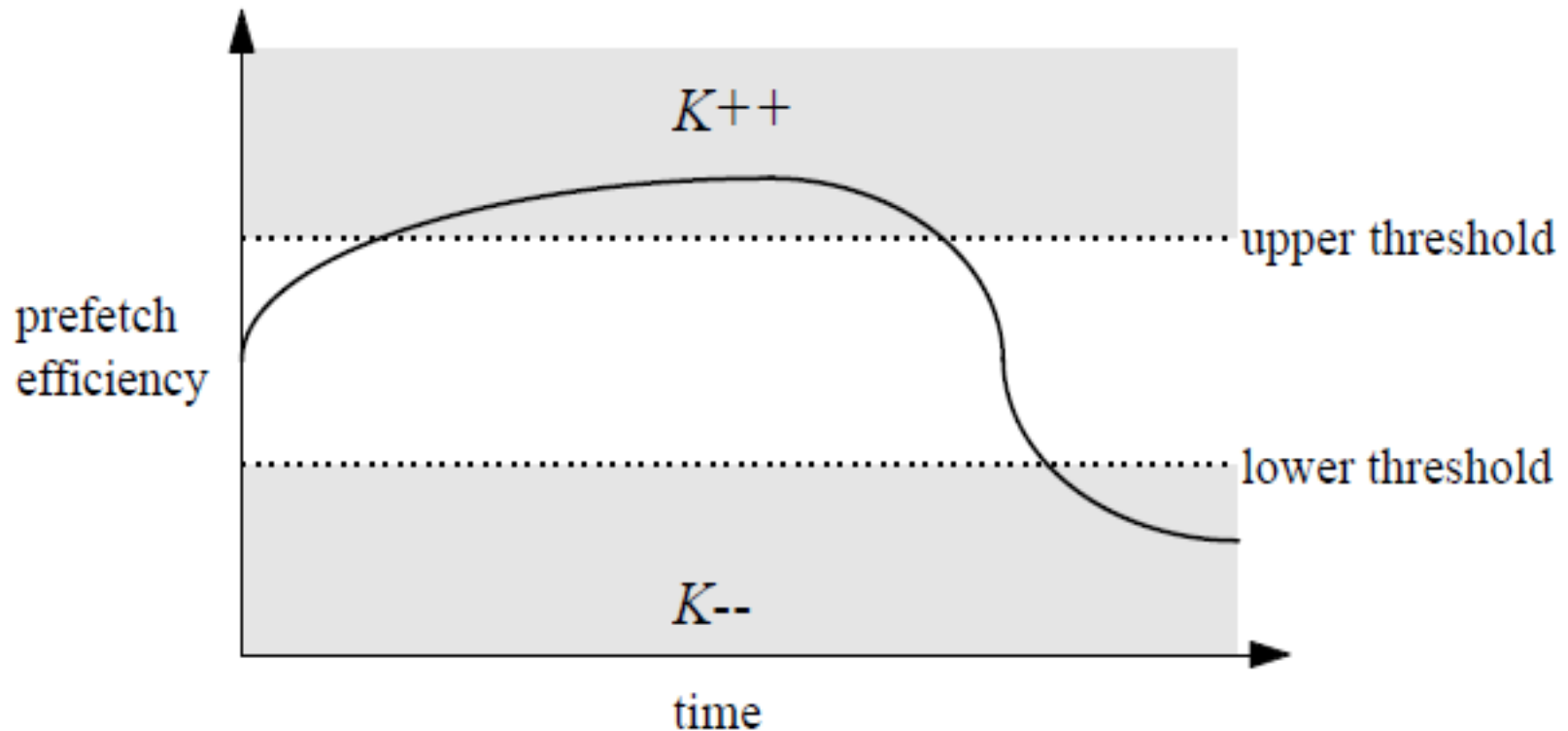
- ❑ Prefetch $K > 1$ subsequent blocks
 - ❑ Additional traffic and cache pollution.

❑ Solution : Adaptive sequential prefetching

- ❑ Vary the value of K during program execution
- ❑ High spatial locality \rightarrow large K value
- ❑ Prefetch efficiency metric periodically calculated
- ❑ Ratio of useful prefetches to total prefetches


- ❑ The value of K is initialized to one, incremented whenever the prefetch efficiency exceeds a predetermined upper threshold and decremented whenever the efficiency drops below a lower threshold
- ❑ If K is reduced to zero, prefetching is disabled and the prefetch hardware begins to monitor how often a cache miss to block b occurs while block $b-1$ is cached
- ❑ Prefetching restarts if the respective ratio of these two numbers exceeds the lower threshold of the prefetch efficiency.

Sequential Adaptive Prefetching



Stride Prefetching

- ❑ *Stride Prefetching* monitors memory access patterns in the processor to detect constant-stride array references originating from loop structures.
- ❑ Accomplished by comparing successive addresses used by memory instructions.
- ❑ Requires the previous address used by a memory instruction to be stored along with the last detected stride, a hardware table called the *Reference Prediction Table (RPT)*, is added to hold the information for the most recently used load instructions.

- 
- ❑ Each RPT entry contains the PC address of the load instruction, the memory address previously accessed by the instruction, a stride value for those entries that have established a stride, and a state field used to control the actual prefetching.
 - ❑ Contains 64 entries; each entry of 64 bits.
 - ❑ Prefetch commands are issued only when a matching stride is detected
 - ❑ However, stride prefetching uses an associative hardware table which is accessed whenever a load instruction is detected.

Pointer Prefetching

- ❑ Effective for pointer intensive programs containing linked data structures.
- ❑ No constant stride.
- ❑ Dependence based prefetching-
 - ❑ Uses two hardware tables.
 - ❑ Correlation table (CT) stores dependence correlation between a load instruction that produces an address (producer) and a subsequent load that uses that address (consumer).
 - ❑ The potential producer window (PPW) records the most recent loaded values and the corresponding instructions. When a load commits, the corresponding correlation is added to CT.

Combined Stride and Pointer Prefetching

- ❑ Objective to evaluate a technique that would work for all types of memory access patterns.
- ❑ Use both array and pointer
- ❑ Better performance
- ❑ All three tables (RPT, PPW, CT)

Hardware vs. Software Approach

Hardware

- ❑ Perform. cost: low
- ❑ Memory traffic: high
- ❑ History-directed
 - ❑ could be less effective
- ❑ Using profiling info.

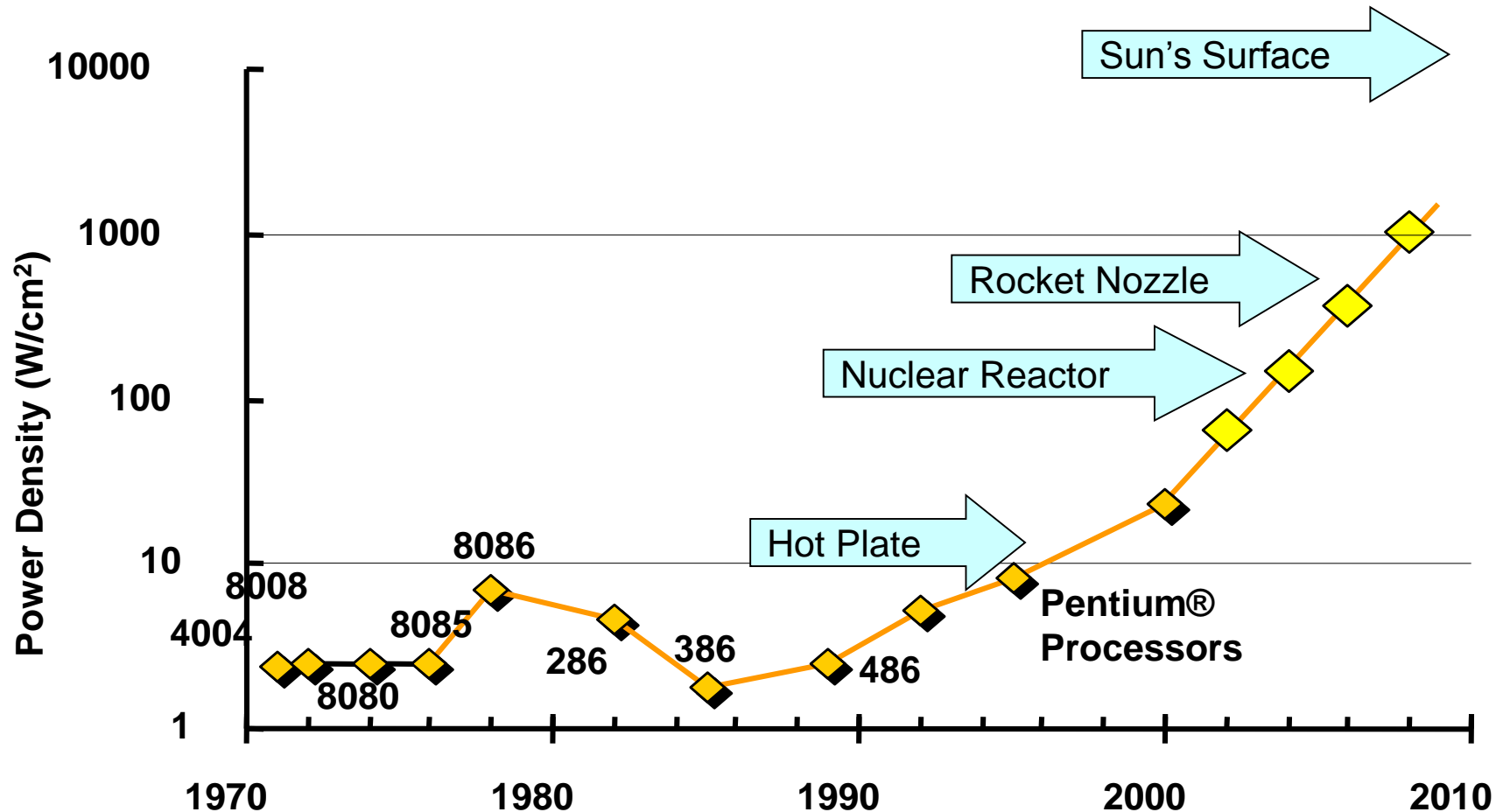
Software

- ❑ Perform. cost: high
- ❑ Memory traffic: low
- ❑ Better Improvement
- ❑ Use human knowledge
 - ❑ inserted by hand

Energy Aware Data Prefetching

- ❑ Energy and power efficiency have become key design objectives in microprocessors, in both embedded and general-purpose domains.
- ❑ Aggressive prefetching techniques often help to improve performance, in most of the applications, but they increase memory system energy consumption by as much as 30%.
- ❑ Power-consumption sources
 - ❑ Prefetching hardware
 - ❑ Prefetch history tables
 - ❑ Control logic
 - ❑ Extra memory accesses
 - ❑ Unnecessary prefetching

Figure shows Power Dissipation in new processors compared with other objects



Source: Intel®

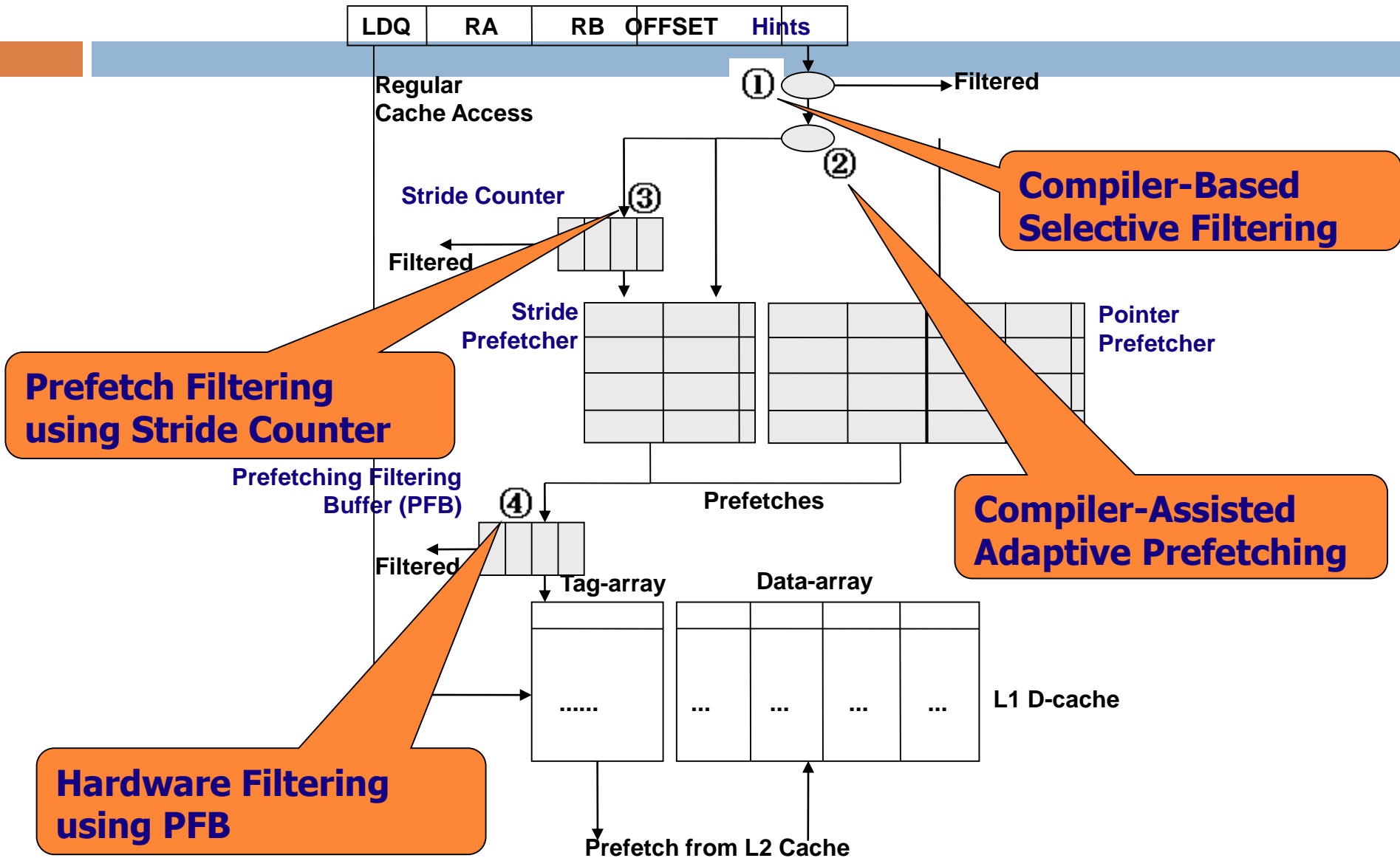
Prefetching Hardware Required

Prefetching Scheme	Hardware Required
Sequential	none
Tagged	1 bit per cache line
Stride	A 64-entry Reference Prediction Table (RPT)
Dependence	A 64-entry Potential Producer Window (PPW) & a 64-entry Correlation Table (CT)
Combined	All three tables (RPT, PPW, CT)

Prefetching Energy Sources

- ❑ Prefetching hardware:
 - ❑ Data (history table) and control logic.
- ❑ Extra tag-checks in L1 cache
 - ❑ When a prefetch hits in L1 (no prefetch needed)
- ❑ Extra memory accesses to L2 Cache
 - ❑ Due to useless prefetches from L2 to L1.
- ❑ Extra off-chip memory accesses
 - ❑ When data cannot be found in the L2 Cache.

Energy-Aware Prefetching Architecture



Energy-Aware Prefetching Technique

- ❑ Compiler-Based Selective Filtering (CBSF)
 - ❑ Only searching the prefetch hardware tables for selective memory instructions identified by the compiler.
- ❑ Compiler-Assisted Adaptive Prefetching (CAAP)
 - ❑ Selectively applying different prefetching schemes depending on predicted access patterns.
- ❑ Compiler-driven Filtering using Stride Counter (SC)
 - ❑ Reducing prefetching energy consumption wasted on memory access patterns with very small strides.
- ❑ Hardware-based Filtering using PFB (PFB)
 - ❑ Further reducing the L1 cache related energy overhead due to prefetching based on locality with prefetching addresses.

Compiler-based Selective Filtering

- ❑ Only searching the prefetch hardware tables for selective memory instructions identified by the compiler.
- ❑ Energy reduced by-
 - ❑ Using loop or recursive type memory access
 - ❑ Use only array and linked data structure memory access

Compiler-Assistive Adaptive Prefetching

- ❑ Select different prefetching scheme based on
 - ❑ Memory access to an array which does not belongs to any larger structure are only fed into the stride prefetcher.
 - ❑ Memory access to an array which belongs to a larger structure are fed into both stride and pointer prefetchers.
 - ❑ Memory access to a linked data structure with no arrays are only fed into the pointer prefetcher.
 - ❑ Memory access to a linked data structure that contains arrays are fed into both prefetchers.

Compiler-Hinted Filtering Using a Runtime SC

- ❑ Reducing prefetching energy consumption wasted on memory access patterns with very small strides.
- ❑ Small strides are not used.
- ❑ Stride can be larger than half the cache line size.
- ❑ Each cache line contain
 - ❑ Program Counter(PC)
 - ❑ Stride counter
- ❑ Counter is used to count how many times the instruction occurs.

Hardware-based Filtering using PFB

- ❑ To reduce the number of L1 tag-checks due to prefetching, we add a PFB to remember the most recently prefetched cache tags.
- ❑ We check the prefetching address against the PFB when a prefetching request is issued by the prefetch engine.
- ❑ If the address is found in the PFB, the prefetching request is dropped and we assume that the data is already in the L1 cache.
- ❑ When the data is not found in the PFB, we perform normal tag lookup and proceed according to the lookup results.
- ❑ The LRU replacement algorithm is used when the PFB is full.

Power Dissipation of Hardware Tables

- ❑ The size of a typical history table is at least 64×64 bits, which is implemented as a fully-associative CAM table.
 - ❑ Each prefetching access consumes more than 12 mW, which is higher than our low-power cache access.
 - ❑ Low-power cache design techniques such as sub-banking don't work.

Conclusion

- ❑ Improve the performance.
- ❑ Reduce the energy overhead of hardware data prefetching.
- ❑ Reduce total energy consumption.
- ❑ Compiler-assisted and hardware-based energy-aware techniques and a new power-aware prefetch engine techniques are used.

References

- ❑ Yao Guo ,”Energy-Efficient Hardware Data Prefetching,” IEEE ,vol.19,no.2,Feb.2011
- ❑ A. J. Smith, “Sequential program prefetching in memory hierarchies,”IEEE Computer, vol. 11, no. 12, pp. 7–21, Dec. 1978.
- ❑ A. Roth, A. Moshovos, and G. S. Sohi, “Dependence based prefetching for linked data structures,” in Proc. ASPLOS-VIII, Oct. 1998, pp.115–126.