UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

{ Memory Hierarchy Optimizations Data Cache Optimizations }

< CHMIELOWSKI Filip, RINGUEDE Delphine >

< CHPC Fall 2024 >

Introduction 1/3

Setting the Stage

Modern computing challenges: high performance, energy efficiency demands (AI, simulations, real-time processing)

Cache misses: delays, increased energy use

Introduction 2/3

Core Objectives

Reduce Latency: Speed up data access

Maximize Hit Rate: Store and retrieve relevant data efficiently

Utilize Memory Effectively: Optimize cache storage to handle workloads

Introduction 3/3

Presentation Focus

Cache behavior, **loop transformations**, and **cache-aware scheduling**, while addressing **challenges** and **emerging trends** of cache optimizations

Beyond class material

Cache Behavior Concepts

Cache Role

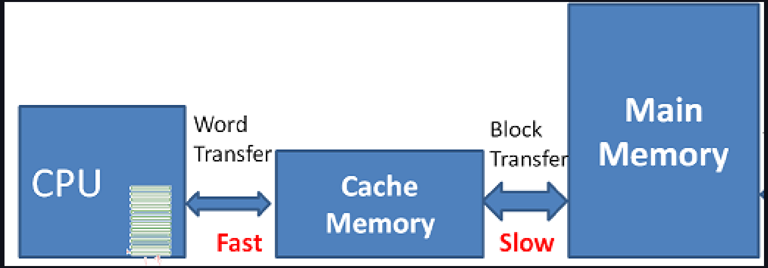
Design Factors

Emerging Challenges

Small, high-speed memory units

Bridge the speed gap between the processor and main memory

Stores frequently or recently accessed data



Cache Behavior Concepts

Cache Role

Design Factors

Emerging Chalenges

Cache efficiency depends on factors such as hierarchy, data replacement and data mapping

Optimizing is challenging due to cache misses, coherence and trade-offs among size, speed and power

Cache Behavior Concepts

Cache Role

Design Factors

Emerging Challenges

Security vulnerabilities

Efficient operation

Understandable cache behavior

Enhanced computational performance and system design

Types of cache misses

Compulsory misses (Cold misses)

Occur when the data is accessed for the first time

Prefetching techniques reduce them by loading the data into the cache before it is actually needed

Capacity misses

Occur when the working set of data is larger than the cache size

Can be solved by optimizing loop structures and limiting the working set size

Conflict misses (Collision misses)

Occur when multiple memory blocks map to the same cache location

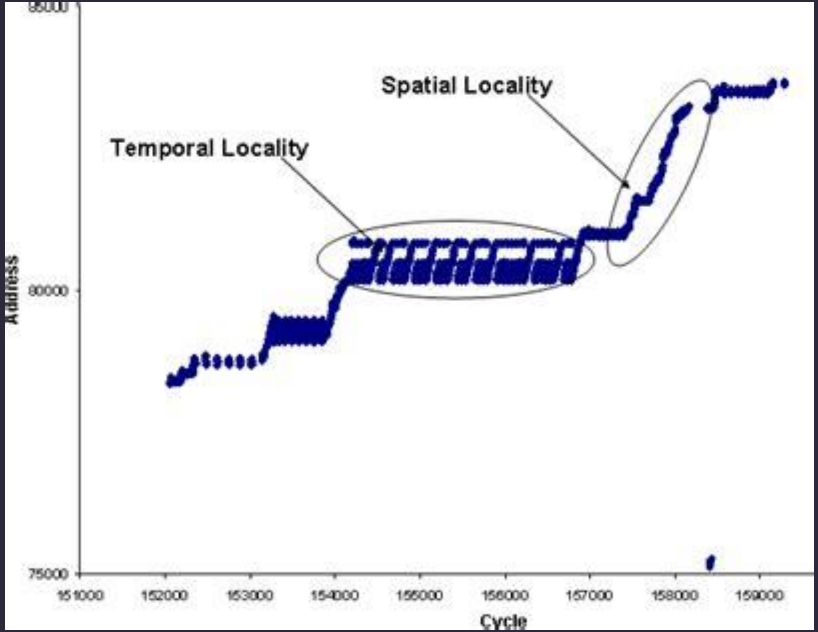
Can be reduced by reorganizing data access patterns or using more associative cache designs

Temporal Locality

The reuse of the same data within a short period of time
Future misses reduction by keeping recently accessed data in cache

Spatial Locality

The access of data elements that are close to each other in memory
This principle works by prefetching nearby data, reducing misses in sequential data accesses



Cache Coherency Challenges

> Frequent updates
> False sharing
> Performance impact

Memory Contention Challenges

> Bus saturation
> NUMA (Non- Uniform Memory Access) delays
> Performance impact

Leveraging Locality & Multicore Efficiency

Cache Optimization Techniques

Cache misses classification
Temporal and spatial locality principles
Tiling

Multicore Challenges

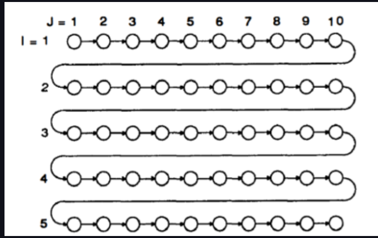
Cache coherency
Memory contention
Tiling
Loop transformations

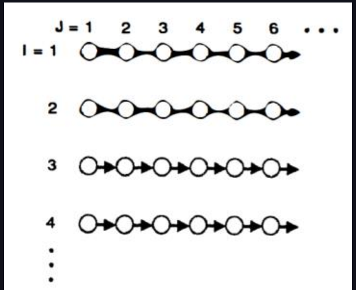
Performance Benefits

Single-core performance boost
Near-linear scalability in multicore systems

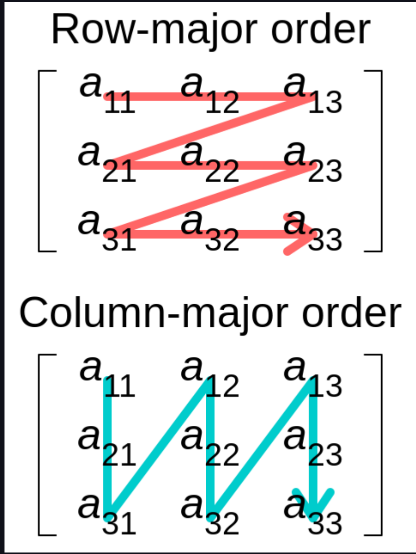
Do you remember these loop transformations? 1/2

Tiling

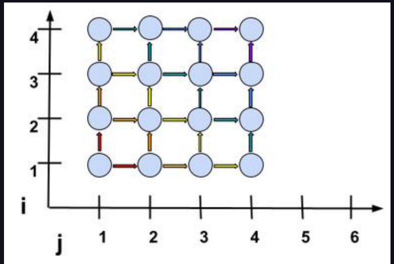


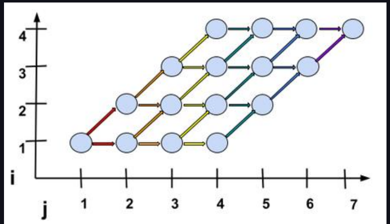


Interchange



Skewing





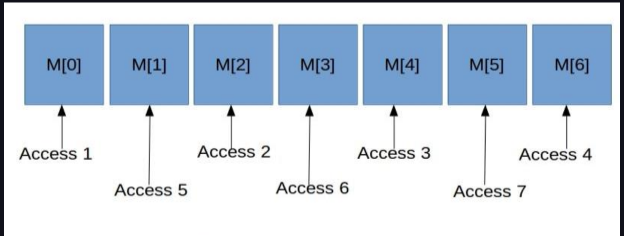
Do you remember these loop transformations? 2/2

Fusion

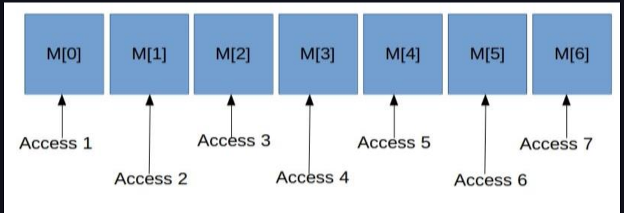
```
int i, a[100], b[100];
for (i = 0; i < 100; i++)  {
        a[i] = 1;
}
for (i = 0; i < 100; i++)  {
        b[i] = 2;
}
```

```
int i, a[100], b[100];
for (i = 0; i < 100; i++) {
        a[i] = 1;
        b[i] = 2;
```

Non-Unit Strides

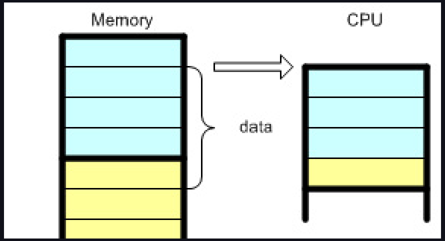


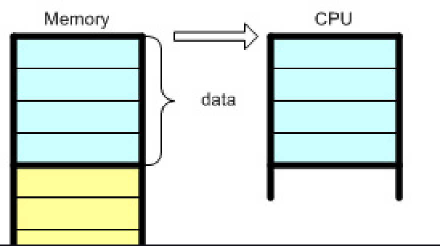
Non unit stride access



Unit stride access

Data Alignment





Loop Transformation Combination

Combining Loop Transformations Considering Caches and Scheduling, E. Wolf (1996)

Loop Transformations	**Fission, fusion, tiling, interchanging, and outer loop unrolling**
Modeling Approach	**Estimates total machine cycle time considering cache misses, software pipelining, register pressure, and loop overhead**
Algorithm	Intelligently searches through possible transformations to select the best overall performance
Performance Improvement	Achieves an average geometric mean improvement of 50% with full optimizations on the MIPS R10000
Compilation Time	**Reasonable**, with modeling and optimizations taking up to 22% of total compile time
Conclusion	Effective and efficient algorithm for optimizing numerical programs, with a few exceptions.

Compiler Support for Loop Transformations

	Flags	Pragmas
Scope	Global (affects the entire codebase)	Localized (specific to code blocks)
Control	compiler-wide setting	Fine-tuned, per-loop or per-function
Example	`-O3` (aggressive optimization)	`#pragma omp parallel` for (parallelism)
Use Case	General performance improvements	Specific optimizations for parallelism or vectorization

Cache-Aware Scheduling Algorithms

01 Cache and Scheduling

Tiling, unrolling and interchange improve cache behavior by keeping data accessible longer

The goal is to minimize performance bottlenecks caused by misses

02 Optimizing Loops for Cache

Tiling creates smaller, cache-friendly blocks

Loop interchange and unrolling enhance data locality and reuse

03 Cache Models in Scheduling

Cache models optimize tile sizes and transformations

The goal is to maximize data reuse and minimize misses for efficient scheduling

Techniques for improving Instruction Level Parallelism

Cache Coherency Fundamentals

Ensures data consistency across multiple processor caches

Main challenges are maintaining consistency with shared data and minimizing coherence traffic

Exploiting Data Locality

Loop Permutation and Fusion: reduce cache line access, minimizing inter-processor communication

Temporal Reuse: reduces the need for frequent cache update

Advanced Techniques

Directory-based Cache Coherency: centralized or distributed directory to track data state

Prefetching with Locality Awareness: pre-load data into local caches to reduce misses

Cache Partitioning: ensures local access to frequently used data

Minimizing False Sharing

Loop Skewing: prevents concurrent access to the same cache line

Data Layout Transformations: avoid cache line sharing between processors

Unroll-And-Jam Transformation

Unrolls outer loops by a specified factor and fuses the unrolled iterations into the inner loop

- ✓ Increased ILP – exposes additional parallelism in the loop body
- ✓ Improved cache utilization – enhances cache locality and reduces cache misses

```
DO J = 1, N
    DO I = 1, M
        A(I, J) = A(I, J - 1) + A(I, J - 2)
    END DO
END DO
```

```
DO J = 1, N, 2
    DO I = 1, M
        A(I, J) = A(I, J - 1) + A(I, J - 2)
        A(I, J + 1) = A(I, J) + A(I, J - 1)
    END DO
END DO
```

Scalar Replacement

Improves performance by replacing repeated memory references with register allocations

- ✓ Reduces memory access costs
- ✓ Lowers cache pressure

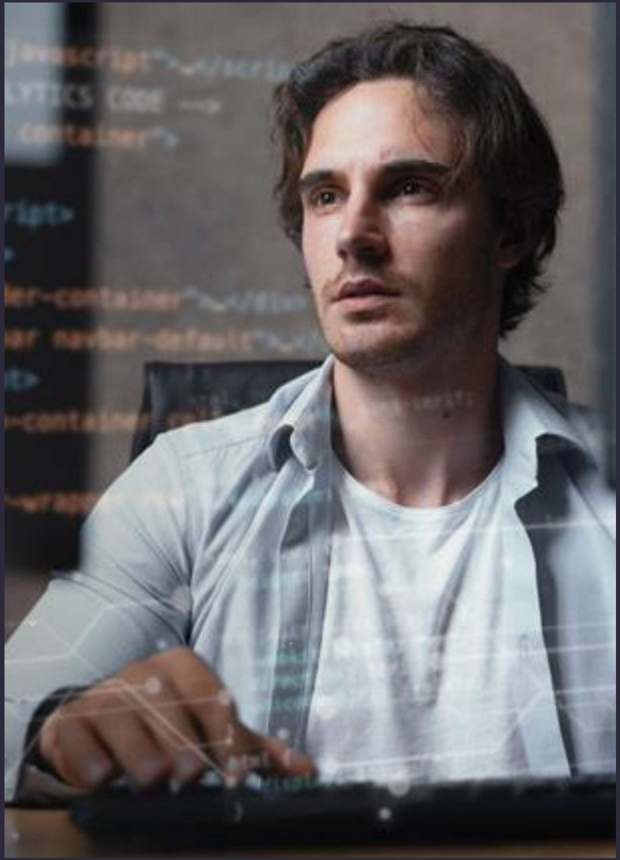
```
A(I, J) = A(I - 1, J) + A(I - 2, J)
```

```
A0 = A(I - 1, J)
A1 = A0 + A(I - 2, J)
A(I, J) = A1
```

Objective of a Cost Function for Data Cache Optimization

Measure and improve the efficiency of memory accesses in a program

Well optimized cost function => reduce memory latency & improve performance



Cost function #1 : RefCost()

Compiler optimizations for improving data locality, Carr 1994

Calculates the access cost for each memory reference based on its access pattern (loop-invariant, unit-stride, or non-unit stride) and the number of iterations in the loop.

$$\textbf{RefCost}(Ref_k, l) = \begin{cases} 1, & \text{if the reference is } \textbf{loop-invariant} \\ \dfrac{\text{trip\_count}_l}{\left(\frac{\text{cls}}{\text{stride}(f_1, i_l, l)}\right)}, & \text{if the reference exhibits } \textbf{unit-stride access} \\ trip\_count_l, & \textbf{otherwise} \text{ (non-unit stride access)} \end{cases}$$

- **trip countl** is the number of iterations for the loop;
- **cls** (cache line size) is the number of memory elements that fit in one cache line;
- **stride(f1, il, l)** is the memory access stride for the reference.

Cost function #2 : LoopCost()

Compiler optimizations for improving data locality, Carr 1994

Quantifies the total memory access cost of a loop nest by summing the costs of all reference groups, adjusted for the trip counts of outer loop.

$$\mathbf{LoopCost}(l) = \sum_{\text{ref\_groups}}^{m} \mathbf{RefCost}(Ref_k(f_1(i_1, ..., i_n), ..., f_j(i_1, ..., i_n)), l)) \times \prod_{h \neq l} trip\_count_h$$

- **ref groups** is the set of reference groups in the loop nest, starting from k = 1
- **trip counth** is the number of iterations for loop h
- **h /= l** ensures that only outer loops are considered for the product

Practical Considerations and Trade-offs

Hardware Constraints | Performance Trade-offs | Scalability & Complexity

Cache hierarchy (L1, L2, L3) requires different optimization strategies for improved data locality

Multi-core and SIMD systems introduce challenges like cache contention and false sharing

Power efficiency must be balanced with performance, using techniques like dynamic voltage scaling

Practical Considerations and Trade-offs

Hardware Constraints

Performance Trade-offs

Scalability & Complexity

Optimizations come with overheads & benefits must outweigh costs

Scaling issues arise in multi-core systems, where cache contention can reduce optimization effectiveness

Practical Considerations and Trade-offs

Hardware Constraints

Performance Trade-offs

Scalability & Complexity

Optimizations must be scalable, considering how memory access patterns change with larger systems

Excessive complexity in optimizations may lead to reduced returns and need careful evaluation

Machine Learning Applications in Cache Optimization

Cache behavior prediction

ML models predict cache accesses and optimize prefetching and replacement, reducing latency & energy use.

Prefetching optimization

ML improves prefetching strategies for spatial patterns and decision trees for workload-specific rules, reducing cache pollution.

Adaptive Cache configuration

ML adapts cache parameters using clustering for workload-specific tuning and online learning for real-time adjustments.

Energy-efficient cache management

ML-driven regression optimizes energy usage while maintaining performance.

Dynamic Cache replacement policies

ML-enhanced policies outperform traditional methods by using supervised learning and neural networks to improve real-time replacement decisions.

Hybrid and Dynamic Optimization Techniques

Hybrid Architectures

Combine SRAM and NVM for performance and energy efficiency.

Dynamic Optimization

Adjust cache parameters and replacement policies in real-time, using machine learning for self-optimization.

CONCLUSION

Data caches address bottlenecks like cache misses, coherency issues, and memory contention, bridging the gap between fast processors and slower memory systems.

Strategies such as tiling, unroll-and-jam and loop fusion improve data locality and minimize latency, enabling efficient memory-intensive computations.

Cache-aware scheduling, coherency protocols and hybrid approaches ensure scalability, energy efficiency, and adaptability to diverse workloads.

Emerging innovations in machine learning-driven cache management, NVM and in-memory processing prepare systems for increasingly complex data demands.

References

- Michael E. Wolf and Monica S. Lam. 1991. **"A data locality optimizing algorithm".** In Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation (PLDI '91). ACM, New York, NY, USA, 30-44.
- Michael E. Wolf, Dror E. Maydan, and Ding-Kai Chen. 1996. **"Combining loop transformations considering caches and scheduling".** In Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture (MICRO 29). IEEE Computer Society, Washington, DC, USA, 274-286.
- Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. 1996. **"Improving data locality with loop transformations".** ACM Trans. Program. Lang. Syst. 18, 4 (July 1996), 424-453.
- Steve Carr and Ken Kennedy. 1994. **"Improving the ratio of memory operations to floating-point operations in loops".** ACM Trans. Program. Lang. Syst. 16, 6 (November 1994), 1768-1810.
- Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. 1994. **"Compiler optimizations for improving data locality".** SIGPLAN Not. 29, 11 (November 1994), 252-262.
- Steve Carr. 1996. **"Combining Optimization for Cache and Instruction-Level Parallelism".** In Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96). IEEE Computer Society, Washington, DC, USA, 238-.

Thank your for your attention!}

< Questions? >