Performance Prediction, Analysis, and Optimization of Numerical Methods on Cache-Based Computer Architectures

Half-day Tutorial
at ISCA 2004 in Munich, Germany

Ulrich Rüde, Markus Kowarschik,
Arndt Bode, Josef Weidendorfer

Overview

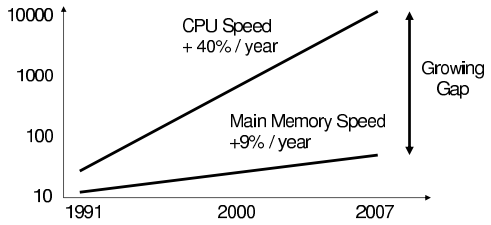
- Introduction and Motivation
- Part I: Cache-Based Architectures
  Modern Microprocessor and Cache Design
- Part II: Performance Analysis and Tools
  Methods, HW Support, Usage
- Part III: Cache Optimizations of Numerical Applications
  Data Layout/Access, Iterative Solvers

Introduction – Contact Us

Ulrich Rüde (ruede@cs.fau.de)
Markus Kowarschik (kowarschik@cs.fau.de)
Arndt Bode (bode@cs.tum.edu)
Josef Weidendorfer (weidendo@cs.tum.edu)

Introduction – Why?

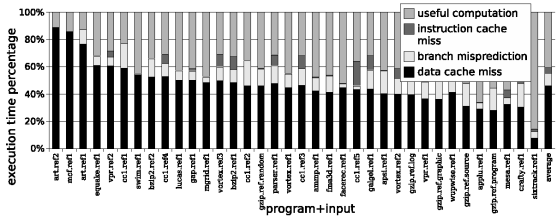
- Bottleneck "Memory Wall" getting worse



Fast Caches Strongly Needed !

Introduction – Why? (cont'd)

- HPC Apps. need to be Cache-Optimized



Reasons for Performance Loss for SPEC2000 on Itanium 1
[Beyls/Hollander, ICCS 2004]

Introduction - Focus

- Architectural Background
- How to find and isolate Bottlenecks
- Strategies for **Sequential** Numerical Code
  (Parallelization should complement Cache-/Sequential Optimizations, not substitute for bad Node Performance)

Our Motivation …

How fast should a solver be?

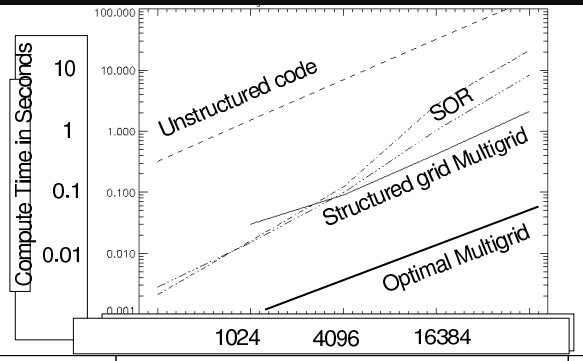
- Poisson problem can be solved by a multigrid method in < 30 operations per unknown (known since late 70ies)
- More general elliptic equations may need O(100) operations per unknown
- A modern CPU can do 1-5 GFLOPS
- So we should be solving 10-50 million unknowns per second
- Should need O(100) Mbytes of memory

How fast are solvers today?

- Often no more than 10,000 to 100,000 unknowns possible before the code breaks
- In a time of minutes to hours
- Needing horrendous amounts of memory

Even state of the art codes
are often very inefficient

Comparison of solvers



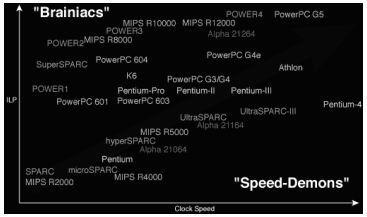
Part I: Cache Based Architectures

CPU Architectures - Overview

- Goal: Understand Performance Issues
- Techniques used
  - in CPUs: Pipelining, Superscalar, OOO, …
  - for Main Memory Chips
  - for Acceleration of Memory Accesses (Caches)

How to Get a fast CPU ?

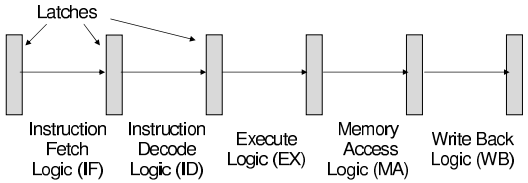
- Increase Clock Rate
- Increase ILP (Instruction Level Parallelism)



http://www.pattosoft.com.au/Articles/ModernMicroprocessors/

Pipelining

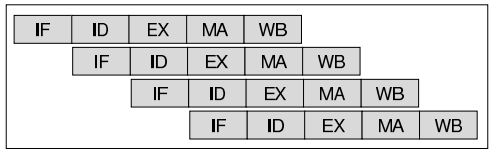
- Pipelined Instruction Execution (RISC)



- Sequential Execution (5 CPI)

Pipelining (Cont'd)

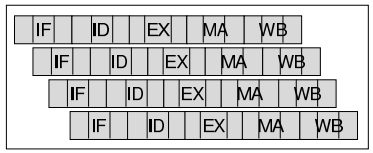
- Pipelined Execution (1 CPI)



  - Slowest Phase/Stage determines Cycle Time
  - All Stages Executed for All Instructions
  - Data / External Dependences causes Stalls

Pipelining (Cont'd)

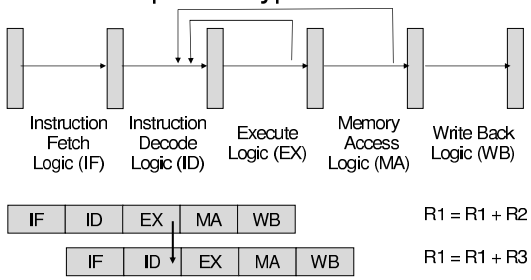
- Superpipelined Execution (1 CPI)



- Deep Pipelines
  - Few Logic/Stage allows High Clock Rates
  - Data Dependencies much more Problematic

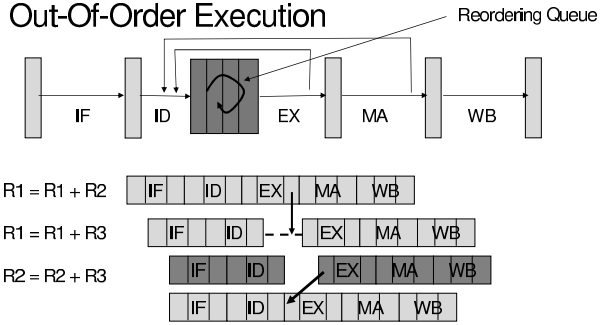
Minimize Data Dependencies

- Introduce Pipeline Bypasses



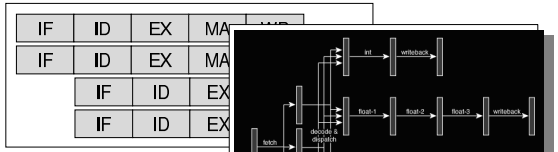
Minimize Data Deps. (Cont'd)

- Out-Of-Order Execution



Pipelining 2

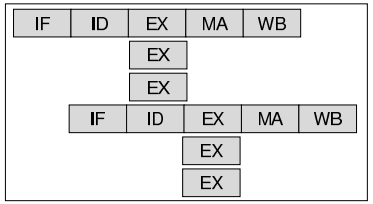
- Superscalar Pipelining (1/2 CPI)



– Reality:
1 IF/ID Stages,
>3 for multiple EX

Pipelining 2 (Cont'd)

- Pipeline with VLIW/EPIC Instructions



– Multiple Parallel Actions per Instruction
– Data Dep. Analysis moved to Compiler

"Ignore" Data/External Dependence

- Speculation, Prediction
  – Needs "Rollback": Shadow Registers
  – Depending on Probability of Multiple Alternatives: Parallel Execution
- For Conditional Jumps: Branch Prediction (Better: Get rid with Predication)
- For Calculations: Value Prediction

Pipelining - Examples

Processor	Pipeline Stages	Superscalar Issue	Reordering	GHz
PowerPC G3/G4	4-7	3	OOO	0.3/1.3
MIPS R10000	5-7	4	OOO	0.195
Alpha 21164	7-9	4	In-Order	0.4
PowerPC G4e	7-12	4	OOO	1.7
Itanium-II	8-10	6 (EPIC)	In-Order	1.3-1.7
UltraSPARC	9	4	In-Order	0.3
Athlon	10-15	3	OOO	1.3 - 2.8
Pentium-II/III	12+	3	OOO	0,3 - 1.0
UltraSPARC-III	14	4	In-Order	1.2
PowerPC G5	16-25	5	OOO	2.0
Pentium-4	20-31	3	OOO	- 3.5

Further CPU Techniques

- Data Parallelism (SIMD Instructions)
  - Large Registers (e.g. 128 Bit), Independent Operations (e.g. Bit0-63, 64-127)
  - Examples
    - SPARC: VIS
    - Alpha: MVI
    - X86: MMX/SSE/SSE2, 3DNow!
    - PowerPC: AltiVec
  - Useful for HPC, Multimedia

Further CPU Techniques (Cont'd)

- SMT (Simulaneous Multithreading)
  - Avoid Pipeline Stalls by Context Switching
  - Tera MTA (128 Contexts, No Cache), Hyper-Threading in Pentium-4 (2 Contexts), UltraSPARC-V, POWER5
- Multi-Core
  - POWER4/5, UltraSPARC-IV, Intel/AMD ?
- Needs Parallelized Software

Summary

- Pipelining Technique good to increase both Clock Rate & ILP
- Problematic (leading to Pipeline Stalls)
  - Data Dependencies
    - Use Bypasses, OOO, Speculation/Prediction
  - External Dependencies (Memory Accesses)
    - Use Speculation (?), Caches (!)
    - Very long Stall Times: – 300 Cycles (3 GHz P-4)

Main Memory

Main Memory – The Problem

- Memory Speed has 2 Sides
  - Bandwidth
  - Latency (Especially worse for Pipelines)
- Main Memory is Off-Chip
  - Fixed limit wire length / Speed of Light
  - High Frequencies are difficult to handle for Board Manufactures
  - Less engineering Progress…

Main Memory - Bandwidth

- Possibilities for Improvement (for both CPU-Chipset, Chipset-Memory)
  - Higher data width (More Bits in Parallel)
  - Higher Bus Clock Rate, Double/Quad-Pumped (e.g. DDR)
  - Memory Bank Interleaving
  - Long Bursts
  - High-Speed Serial Links (HyperLink, Rambus) easier for off-chip

Main Memory – Bandwidth (Cont'd)

- Example on Improvement:
  - 100 MHz Mem. Bus, 64 Bit, SDRAM (PC100): max. 800 MB/s (ca. 1997)
  - 200 MHz Mem. Bus, 128 Bit (2 Banks), DDR 400: max. 6,4 GB/s (ca. 2003)

Main Memory Speed - Latency

- Latency for SDRAM typically 9 Cycles (at Memory Bus Clock Rate!)
  - 1 from CPU to Chipset
  - 1 from Chipset to RAM Chip
  - 2 for RAS-to-CAS Delay (for Page Miss)
  - 2 for CAS Latency
  - 1 to get data to Output Buffer of RAM Chip
  - 1 from RAM Chip to Chipset
  - 1 from Chipset to CPU

Main Memory – Latency (cont'd)

- Assume 133 MHz Memory Bus Speed
  1.3 GHz CPU: 90 Cycles (3 GHz: 207 Cycles)
- This does not improve much with DDR/DDR-II
- Getting worse with SMP machines
- Possibility for Improvement
  - Reduce Latency On-Board
    - High-Speed CPU-Chipset (200MHz QDR P-4)
    - Memory Controllor in CPU (UltraSparc, Opteron)

The Memory Wall

- Bandwidth Needs of CPUs
  - 4 FP Load/Cycle (2x: $x_i = x_i + a_i y_i$) with 1,5 GHz: 48 GB/s (Itanium-II)
  - At best without Latency at all
- Memory can deliver
  - Max. 6,4 GB/s (Itanium-II)
  - Best Latency ~ 66 ns = 121 Cyc. at 1,8GHz (Opteron)
- Gap will be growing

The Memory Wall - Solution

- Caches
  - Small but Fast on-board Buffers holding copies of Main Memory Content
  - Requirements
    - High Bandwidth: Data Paths with large Width ("No Problem" on-chip)
    - Small Latency: Tightly coupled to Registers (Better use Cache Hierarchies!)
  - Only works when Cache Copies are Reused!

Cache Design

Caches Work when…

- Memory Access Stream of an Application shows some Locality ("Principles of Locality")
  - Temporal locality: an item referenced now will be referenced again soon
  - Spatial locality :an item referenced now indicates that neighbors will be referenced soon

Basic Cache Properties

- Cache work on Copies of Memory Blocks
  - Called *Cache lines*
  - Size typically 32 - 128 bytes, based on Optimum for Data Bus and Memory Design
  - Transfers to Main Memory always at Block Granularity
  - Blocks exploit Spatial Locality
  - Each Block knows its Memory Address (Tag)

Basic Cache Properties (Cont'd)

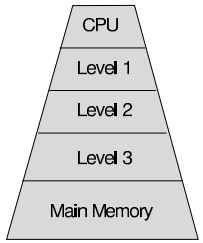
- Where gets a copy of a given memory block placed ?
  - In every Cache Line: **Full Associativity**
    - Best, but very costly
    - Needs a compare for every line and access
  - In *N* Cache Lines: **N-Way Associativity**
    - Cache partitioned in Sets with *N* Lines each
  - Special case *N*=1: **Direct Mapped**
  - For N-Way/DM: Eviction of lines possible even if Cache is not Full (Conflict Case)

Basic Cache Properties (Cont'd)

- Requests are satisfied from a cache
  - if it holds the appropriate copy: **Cache Hit**
  - if data is not in cache: **Cache Miss**
- When Cache is full on a Miss
  - Which Cache Line to replace? **Replacement Policy**
    - Theor. best: Longest Unused Copy (MIN)
    - Often Used: LRU (Least Recently Used)
    - Others: LFU (Least Frequently Used), RANDOM
    - Promising are Adaptive Policies based on Locality Behavior
  - Not applicable to Direct-Mapped

Basic Cache Properties (Cont'd)

- Multiple Levels: L1, L2, sometimes L3



Size	Lat.	Bandwidth
16 kB	1	48 GB/s
256 kB	5+	6,4 GB/s
3 MB	12+	6,4 GB/s
	>100	6,4 GB/s

(Itanium-II)

Basic Cache Properties (Cont'd)

- On a Write Access:
  - Create Copy in Cache (Allocation Policy)?
  - If Yes (Write Policy):
    - Also update slower levels/main memory: **Write Through**
      - Constant high Bus Traffic
    - Keep modified Memory Line in Cache, write at eviction time: **Write Back**
      - Needs a "Dirty" Bit per Cache Line
      - Less Bus Traffic, Risk of "Write Storms"

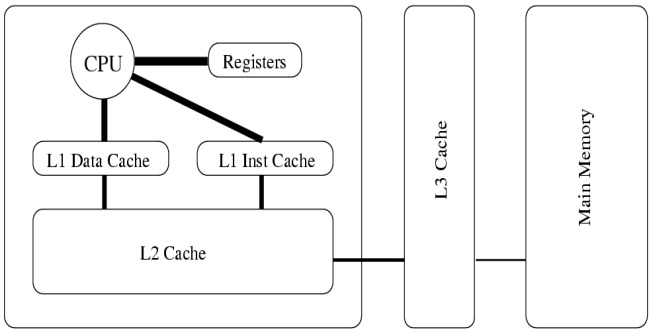
Basic Cache Properties (Cont'd)

- Is a Cache Line accessed via
  - Virtual Address (per OS Task/Address-Space)
    - Fast, no MMU Translation needed before Access
    - Flush needed on Task Switch, Consistency Problems on SMP
    - Typically used for L1
  - Pyhsical Address
    - Needs Translation from virtual Address (supported via TLB Cache!)
  - Hybrid (Virtually Indexed, Physically tagged)

Cache Configurations

- Separate (L1)-Caches
  - Instructions
  - Data
  - Page Table Translations for MMU
    - To map virtual to phys. addresses
    - Called TLB (Translation Lookaside Buffer)
    - Often Full-Associative, 2 Levels
    - Very expensive
- L2/L3 “unified”

Cache Configurations (Cont’d)



Cache Issues

- Transparency for User/Programmer
- Data consistency with main memory and other Caches on SMP machines
  - Cache Consistency Protocol
  - On SMP: MESI (via Bus Snooping)
  - On NUMA: Directory Based (Node Lists for every Cache Line)

Effect of Cache Hit Ratio

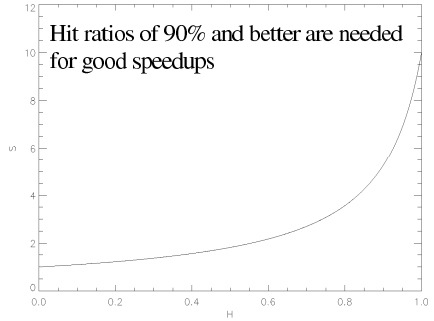
The cache efficiency is characterized by the cache hit ratio, the *effective* time for a data access is

$$T_{\text{eff}} = H \cdot T_c + (1 - H) \cdot T_m.$$

The *speedup* is then given by

$$S = \frac{T_m}{T_{\text{eff}}} = \frac{1}{1 - H(1 - T_c/T_m)}$$

Cache effectiveness depends on the hit ratio



Summary: Cache Organization

- Number of Cache Levels
- Set Associativity
- Physical/Virtual/Hybrid Addressing
- Write-through/write-back Policy
- Replacement Policy
- Cache Line Size

Cache Organization (Cont'd)

Processor	Size [kB] L1D/L2	Ass. L1D/L2	Line Size	L1 Hit Latency
Pentium-II/III	16	4	64	3
UltraSPARC	16	1	32	2
Itanium I	16/96	4/6	64	2
Itanium II	16/256/3MB	4	64	1
Alpha 21264	64/4MB	2/1	32/64	2
Pentium-4	8/256	4/8	64	1(3)
Athlon	64/256	2	64	3
PowerPC 3	64/4MB	128/1	128	2
PowerPC 4	32/1.5MB	8/2	128	3

Cache Prefetching

- Memory Latency not bad if hidden by Computation (overlapping)
- Idea: Use Memory Bus when it would be idle (CPU is working/using data from L1/2)
  - Prefetch Data into L2 which is used somewhat in the Future
  - Software Prefetching: Programmer Hints
    - Which Address, How used (only once/multiple times)
    - Never raises Exceptions
  - Hardware Prefetch: Predict Future from recent access stream of Application

Cache Prefetching (Cont'd)

- HW-Prefetching
  - Stop Prefetching at VM Page Boundaries
  - Algorithms
    - Often used: Stream Up/Down Detection Pentium-4 (8) ,P-M (16)
    - Stride Detection (indexed by Instruction Address)
    - Address Correlation Predictors e.g. for linked Lists (Global Detection of Address Diff. Patterns)
  - Can improve Performance Dramatically

References Part I

- Jason Patterson: Modern Microprocessors 90 Minute Guide (http://www.pattosoft.com.au/Articles/ModernMicroprocessors)
- Microprocessor Report (http://www.mdronline.com/mpr)
- Ars Technica (http://arstechnica.com)
- Intel, AMD, IBM: Processor Manuals

Part II – Analysis/Tools

(for Sequential Code)

Analysis/Tools - Overview

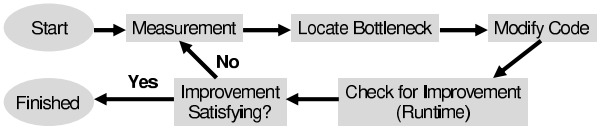
- Basics and Methods
- Hardware Support
- Profiling tools

Performance Analysis

- Why to use Performance Analysis
  - Locate Code Regions (where most Time is spent in) for Optimizations to get Peak Performance
  - Check Correctness of Assumptions on Runtime Behavior
  - Chose Best Algorithm from Alternatives for a Problem
  - Get Knowledge about unknown Code

Performance Analysis (Cont'd)

- How to do Performance Analysis
  - At End of (fully tested) Implementation
  - On Compiler-Optimized Release Version
  - With typical/representative Input Data
  - Steps of Optimization Cycle



Performance Analysis (Cont'd)

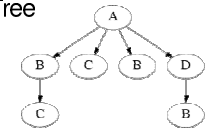
- Performance Bottlenecks (sequential)
  - Logical Errors: Too often called Functions
  - Algorithms with bad Complexity or Implementation
  - Bad Memory Access Behavior (Bad Layout, Low Locality)
  - Lots of (conditional) Jumps, Lots of (unnecessary) Data Dependencies, ...

Performance Measurement

- Wanted:
  - Time Partitioning with
    - Reason for Performance Loss (Stall because of…)
    - Detailed Relation to Source (Code, Data Structure)
    - Suitable Metrics to get Hints for Improvements
  - Runtime Numbers
    - Call Relationships, Call Numbers
    - Loop Iterations, Jump Counts
  - No Perturbation of Results b/o Measurement

Measurement - Terms

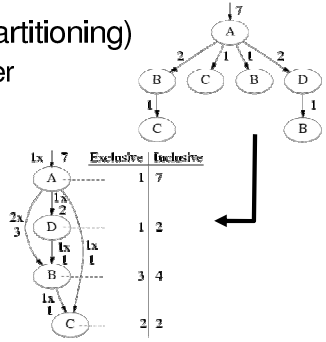
- Trace
  - Record of Time-Stamped Event Stream
    - Enter/Leave of Code Region, Actions, …
      Example: Dynamic Call Tree



    - Huge Amount of Data (Linear to Runtime)
    - Unneeded for Sequential Analysis (?)

Measurement – Terms (Cont'd)

- Profiling (e.g.Time Partitioning)
  - Record Summary over Execution
    - Exclusive, Inclusive Cost / Time, Counters
    - Example: DCT → DCG (Dynamic Call Graph)
  - Amount of Data Linear to Code Size



Methods

- Precise Measurements
  - Increment Counter (Array) on Event
  - Attribute Counters to
    - Code (Context, Basic Block Sequence)
    - Data (Adress, Access Pattern)
  - Data Reduction Possibilites
    - Selection (Event Type, Code Range, Data)
    - Online Processing (Compression, …)
  - Needs Instrumentation (Measurement Code)

Methods - Instrumentation

- Manual
- Source Instrumentation
- Library Version with Instrumentation
- Compiler
- Binary Editing
- Runtime Instrumentation / Compiler
- Runtime Injection

Methods (Cont'd)

- Statistical Measurement ("Sampling")
  - TBS (Time Based), EBS (Event Based)
  - Assumption:
    Event Distribution over Code Approximated by checking every N-th Event only
  - Similar Way for Iterative Code:
    Measure only every N-th Iteration
- Data Reduction Tunable
  - Compromise between Quality/Overhead

Methods (Cont'd)

- Simulation
  - Events for (not existant) HW Models
  - Results not influenced by Measurement
  - Compromise Quality / Slowdown
    - Rough Model = High Discrepancy to Reality
    - Detailed Model = Best Match to Reality
      But: Reality (CPU) often unknown…
  - Comparison of Models with Changes of Architecture Parameters possible

Hardware Support

- Monitor Hardware
  - Event Sensors (in CPU, on Board)
  - Event Processing / Collection / Storing
    - Best: Separate HW
    - Comprimise: Use Same Resources after Data Reduction
  - Most CPUs nowadays include Performance Counters

Performance Counters

- Multiple Event Sensors
  - ALU Utilisation, Branch Prediction, Cache Events (L1/L2/TLB), Bus Utilisation
- Processing Hareware
  - Filter Techniques (Itanium, Pentium-4)
    - Opcode, Code/Data Range, Thresholds (Latency!)
  - Counter Registers
    - Itanium2: 4, Pentium-4: 18, Opteron: 8
      Athlon: 4, Pentium-II/III/M: 2, Alpha 21164: 3

Performance Counters (Cont'd)

- Two Uses:
  - Read
    - Get Precise Count of Events in Code Regions by Enter/Leave Instrumentation
  - Interrupt on Overflow
    - Allows Statistical Sampling
    - Handler Gets Process State & Restarts Counter
- Both can have Overhead
- Often Difficult to Understand

Performance Counters (Cont'd)

- Common Misconceptions
  - Don't take absolute values for granted!
    - Instruction Counts: Including Speculation?
    - Miss Counts: influenced by OOO, including Prefetch Events (?), Nothing about Stall Time
- Example (on Pentium-4, in C)
  - Iteration over 8 x 1000000 Matrix
    - Row-First 1/8 Miss Count of Column-First
    - Still 100% Slower! (HW-Prefetch, Loop Overhead)

Tools - Measurement

- Read Hardware Performance Counters
  - Specific: PerfCtr (x86), Pfmon (Itanium), perfex (SGI), DCPI (Alpha)
  - Portable: PAPI, PCL
- Statistical Sampling
  - PAPI, DCPI, Pfmon (Itanium), OProfile (Linux), VTune (commercial - Intel), Prof/GProf (TBS)
- Instrumentation
  - GProf, Pixie (HP/SGI), Atom (Alpha), VTune (Intel)
  - DynaProf (Using DynInst), Valgrind (x86 – Simulation)

Tools – Example 1

- GProf (Compiler generated Instr.):
  - Function Entries increment Call Counter for (caller, called)-Tupel
  - Combined with Time Based Sampling
  - Compile with "gcc – pg ..."
  - Run creates "gmon.out"
  - Analyse with "gprof ..."
  - Overhead still around 100% !
- Available with GCC on UNIX

Tools – Example 2

- PCL Hardware performance monitor: hpm
- Run:

```
hpm --events PCL_CYCLES, PCL_MFLOPS prog

hpm: elapsed time: 5.172 s
hpm: counter 0   : 2564941490 PCL_CYCLES
hpm: counter 1   : 19.635955 PCL_MFLOPS
```

(Similar: Perfex, Pfmon, hpccount)

Tools – Example 2 (Cont'd)

```
include <pcl.h>

int main(int argc, char **argv) {
    // Initialization
    PCL_CNT_TYPE i_result[2];
    PCL_FP_CNT_TYPE fp_result[2];
    int counter_list[]= {PCL_FP_INSTR, PCL_MFLOPS},res;
    unsigned int flags= PCL_MODE_USER;
    PCL_DESCR_TYPE descr;
    PCLinit(&descr);
    if(PCLquery(descr,counter_list,2,flags)!= PCL_SUCCESS)
          // Issue error message …
    else {
          PCL_start(descr, counter_list, 2, flags);
          // Do computational work here …
          PCLstop(descr,i_result,fp_result,2);
          printf("%i fp instructions, MFLOPS: %f\n",
                  i_result[0], fp_result[1]);}
          PCLexit(descr);
    }
    return 0;
}
```

Event Types to Measure

Start

Stop

Similar: PAPI, Pflib (Itanium)

Tools – Example 3

- DCPI
  - Start the DCPI daemon (dcpid)
  - Run your code
  - Stop the DCPI daemon
  - Use DCPI tools to analyze the profiling data
    - **dcpiprof**: Breakdown of CPU time by procedures
    - **dcpilist**: Source Annotation
- Similar: OProfile, VTune

Tools – Example 3 (Cont'd)

```
dcpiprof:

   dmiss     self%   cum%     procedure       image
   33320    72.84% 72.84%     mgSmooth        ./mg
   10008    21.88% 94.72%     mgRestriction   ./mg
    2411     5.27% 99.99%     mgProlongCorr   ./mg

dcpiwhatcg:

   I-cache (not ITB)   0.1% to  7.4%
   D-cache miss       24.2% to 27.6%
   DTB miss           53.3% to 57.7%
   ..
   Useful              7.9%
```

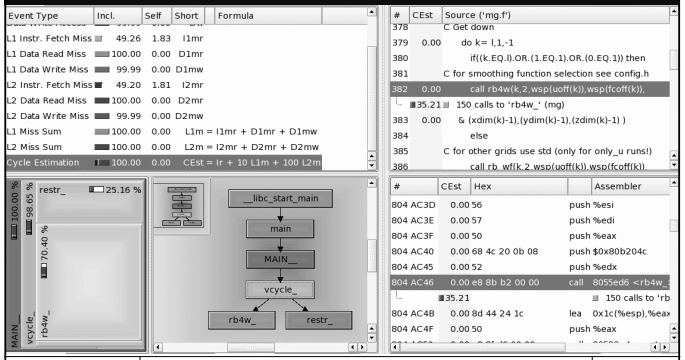
Tools – Example 4

- Calltree (Linux/x86), GPL
  - Cache Simulator (+ HW Pf.) using Valgrind
  - Builds up Dynamic Call Graph
  - Comfortable Runtime Instrumentation
    - Run with “calltree prog”
    - Generates “cachegrind.out.xxx”
    - Results with “ct_annotate” or “kcachegrind” (GUI)
  - Short Demo at End of Tutorial
  - http://kcachegrind.sf.net

Tools - Visualization

- Wanted:
  - Zoomability from Overview to Details
  - Easy Graphical Visualizations
  - GUI enabling fast Browsing
  - Ordered Lists, Source Annotation, Call Graphs
- Text: Most Tools
- GUI: VTune, HPCView, Vampir, TAU, Perfometer, KCachegrind

Visualization – Example KCachegrind (Short Demo at End of Tutorial)



Part III - Applications

Basic efficiency guidelines

- Choose the best algorithm
- Use efficient libraries
- Find good compiler and options
- Use suitable data layout

Choose the best algorithm

Example: Solution of linear systems arising from the discretization of a special PDE

- Gaussian elimination (standard): $n^3/3$ ops
- Banded Gaussian elimination: $2n^2$ ops
- SOR method: $10n^{1.5}$ ops
- Multigrid method: 30n ops

Choose the best algorithm (cont'd)

- For n large, the multigrid method will always outperform the others, even if it is badly implemented
- Frequently, however, two methods have approximately the same complexity, and then the better implemented one will win

Use efficient libraries

Good libraries often outperform own software

- Clever, sophisticated algorithms
- Optimized for target machine
- Machine-specific implementation

Sources for libraries

- Vendor-independent
  - Commercial: NAG, IMSL, etc.; only available as binary, often optimized for specific platform
  - Free codes: e.g., NETLIB (LAPACK, ODEPACK, etc.), usually as source code, not specifically optimized
- Vendor-specific; e.g., cxml for Compaq/Alpha with highly tuned LAPACK routines, for example

Sources for libraries (cont'd)

- Many libraries are quasi-standards
  - BLAS
  - LAPACK
  - etc.
- Parallel libraries for supercomputers
- Specialists can sometimes outperform vendor-specific libraries

Find good compiler options

- Modern compilers have numerous flags to select individual optimization options
  - **`-On`**: successively more aggressive optimizations, n=1,...,5
  - **`-fast`**: may change round-off behavior
  - **`-unroll`**
  - **`-arch`**
  - Etc.
- Learning about your compiler is usually worth it: RTFM

Find good compiler options (cont'd)

Hints:

- Read **`man cc`** (**`man f77`**)
- Look up compiler options documented in www.specbench.org for specific platform
- Experiment and compare performance

Use suitable data layout

Access memory in order! In C/C++, for a 2D matrix

```
double a[n][m];
```

the loops should be such that

```
for (i...)
  for (j...)
    a[i][j]...
```

In FORTRAN, it must be the other way round

Apply *loop interchange*, if necessary, see below

Use suitable data layout (cont'd)

Other example: *array merging*

Three vectors accessed together (in C/C++):

```
double a[n],b[n],c[n];
```

can often be handled more efficiently by using

```
double abc[n][3];
```

In FORTRAN again indices permuted

Optimization Techniques

How to make codes fast

1. Use a fast algorithm (e.g., multigrid)
   I. It does not make sense to optimize a bad algorithm
   II. However, sometimes a fairly simple algorithm that is well implemented will beat a very sophisticated, super method that is poorly programmed
2. Use good coding practices
3. Use good data structures
4. Apply appropriate optimization techniques

Optimization of FP operations

- Loop unrolling
- Fused Multiply-Add (FMA) instructions
- Exposing instruction-level parallelism (ILP)
- Software pipelining (again: exploit ILP)
- Aliasing
- Special functions
- Eliminating overheads
  - if statements
  - Loop overhead
  - Subroutine calling overhead

Loop unrolling

- Simplest effect of loop unrolling: fewer test/jump instructions (fatter loop body, less loop overhead)
- Fewer loads per flop
- May lead to threaded code that uses multiple FP units concurrently (instruction-level parallelism)

- How are loops handled that have a trip count which is not a multiple of the unrolling factor?
- Already fat loops do hardly benefit from unrolling (instruction cache capacity!)
- Very short loops may suffer from unrolling or benefit strongly

Loop unrolling: Making fatter loop bodies

Example: DAXPY operation

```
do i=1,N
  a(i)= a(i)+b(i)*c
enddo
```

```
ii= imod(N,4)
do i= 1,ii
  a(i)= a(i)+b(i)*c
enddo
do i= 1+ii,N,4
  a(i)= a(i)+b(i)*c
  a(i+1)= a(i+1)+b(i+1)*c
  a(i+2)= a(i+2)+b(i+2)*c
  a(i+3)= a(i+3)+b(i+3)*c
enddo
```

```
do i= 1,N,4
  a(i)= a(i)+b(i)*c
  a(i+1)= a(i+1)+b(i+1)*c
  a(i+2)= a(i+2)+b(i+2)*c
  a(i+3)= a(i+3)+b(i+3)*c
enddo
```

Preconditioning loop handles cases when N is no multiple of 4

Loop unrolling: Improving flop/load ratio

Analysis of the flop-to-load-ratio often unveils another benefit of unrolling:

```
do i= 1,N
  do j= 1,M
    y(i)=y(i)+a(j,i)*x(j)
  enddo
enddo
```

Innermost loop: two loads and two flops performed; i.e., we have one load per flop

Loop unrolling: Improving flop/load ratio

```
do i= 1,N,2
  t1= 0
  t2= 0
  do j= 1,M,2
    t1= t1+a(j,i)   *x(j) +a(j+1,i)   *x(j+1)
    t2= t2+a(j,i+1) *x(j) +a(j+1,i+1) *x(j+1)
  enddo
  y(i)  = t1
  y(i+1)= t2
enddo
```

Both loops unrolled twice

Innermost loop: 8 loads and 8 flops!

Exposes instruction-level parallelism

How about unrolling by 4?

Watch register spill!

Fused Multiply-Add (FMA)

On many CPUs (e.g., IBM Power3/Power4) there is an instruction which multiplies two operands and adds the result to a third

Consider code

```
a= b + c*d + f*g
```

versus

```
a= c*d + f*g + b
```

Can reordering be done automatically?

Exposing ILP

```
program nrm1
real a(n)
tt= 0d0

do j= 1,n
  tt= tt + a(j) * a(j)
enddo

print *,tt
end
```

```
program nrm2
real a(n)
tt1= 0d0
tt2= 0d0
do j= 1,n,2
  tt1= tt1 + a(j)*a(j)
  tt2= tt2 + a(j+1)*a(j+1)
enddo

tt= tt1 + tt2
print *, tt
end
```


Exposing ILP (cont'd)

- Superscalar CPUs have a high degree of on-chip parallelism that should be exploited
- The optimized code uses temporary variables to indicate independent instruction streams
- This is more than just loop unrolling!
- Can this be done automatically?
- Change in rounding errors?

Software pipelining

- Arranging instructions in groups that can be executed together in one cycle
- Again, the idea is to exploit instruction-level parallelism (on-chip parallelism)
- Often done by optimizing compilers, but not always successfully
- Closely related to loop unrolling
- Less important on out-of-order CPUs

Aliasing

- Arrays (or other data) that refer to the same memory locations
- Aliasing rules are different for various programming languages; e.g.,
  - FORTRAN forbids aliasing, unspec. result
  - C/C++ permit aliasing
- This is one reason why FORTRAN compilers often produce faster code than C/C++ compilers do

Aliasing (cont'd)

Example:

```
subroutine sub(n, a, b, c, sum)
double precision sum, a(n), b(n), c(n)

sum= 0d0
do i= 1,n
  a(i)= b(i) + 2.0d0*c(i)
  sum = sum + b(i)
enddo
```

FORTRAN rule: two variables cannot be aliased, when one or both of them are modified in the subroutine

Correct call: `call sub(n,a,b,c,sum)`

Incorrect call: `call sub(n,a,a,c,sum)`

Why should aliasing be forbidden?

```
tb = b(1)
tc = c(1)
do i=1,n-1
   ta = tb+2.d0*tc
   tc = c(i+1)

   sum = sum + tb
   tb = b(i+1)

   a(i) = ta
enddo
i = n
ta = tb + 2.0 *tc
sum = sum +tb
a(i) = ta
```

Assume that each block of ops can be executed in 1 cycle
Latency 1 cycle for load
Latency 2 cycles for a flop

Programm produces wrong results when used with second call a=b

Why should aliasing be forbidden? (cont'd)

```
do i=1,n
   LOAD tc = c(i)
   LOAD tb = b(i)
   ta = tb * 2d0 * tc
   NOOP
   STORE a(i) = ta
   LOAD sum
   LOAD tb = b(i)
   sum = sum + tb
   STORE sum
enddo
```

Consider version correct with aliasing

NOOP indicates CPU waiting because of dependencies

Explicit LOAD/STORE

Couldn't we save LOAD/STORE of sum?

Aliasing (cont'd)

- Aliasing forbidden in FORTRAN (result undefined when used)
- Aliasing is legal in C/C++: compiler must produce conservative code
- More complicated aliasing is possible; e.g., `a(i)` with `a(i+2)`
- C/C++ keyword `restrict` or compiler option `-noalias`

Special functions

- / (divide)
- sqrt
- exp, log
- sin, cos, …
- etc.

are expensive (up to several dozen cycles)

- Use math. identities; e.g., log(x) + log(y) = log(x*y)
- Use special libraries that
  - vectorize when many of the same functions must be evaluated
  - trade accuracy for speed, when appropriate

Eliminating overheads: `if` statements

`if` statements …

- Prohibit some optimizations (e.g., loop unrolling in some cases)
- Evaluating the condition expression takes time
- CPU pipeline may be interrupted
- (dynamic jump prediction)

Goal: avoid `if` statements in the innermost loops

No generally applicable technique exists ☹

Eliminating `if` statements – Example

```
subroutine
  thresh0(n,a,threshh,ic)
dimension a(n)
ic= 0
tt= 0.d0
do j= 1,n
tt= tt + a(j) * a(j)
if (sqrt(tt).ge.thresh) then
  ic= j
  return
endif
enddo
return
end
```

- Avoid `sqrt` in condition!
- Add `tt` in blocks of 128 for example (without condition) and repeat last block when condition is violated

Eliminating loop overheads

- For starting a loop, the CPU must free certain registers: loop counter, address, etc.
- This may be significant for a short loop!
- Example: for n>m

```
do i= 1,n
do j= 1,m
...
```

is less efficient than

```
do j= 1,m
do i= 1,n
...
```

- However, data access optimizations are even more important, see below

Eliminating subroutine calling overhead

- Subroutines (functions) are very important for structured, modular programming
- Subroutine calls are expensive (on the order of up to 100 cycles)
- Passing value arguments (copying data) can be extremely expensive, when used inappropriately
- Passing reference arguments (as in FORTRAN) may be dangerous from a point of view of correct software
- Reference arguments (as in C++) with `const` declaration
- Generally, in tight loops, no subroutine calls should be used

Eliminating subroutine calling overhead (cont'd)

- Inlining: `inline` declaration in C++ (see below), or done automatically by the compiler
- Macros in C or any other language

```
#define sqre(a) (a)*(a)
```

What can go wrong:

`sqre(x+y)` → `x+y*x+y`

`sqre(f(x))` → `f(x) * f(x)`

What if f has side effects?

What if f has no side effects, but the compiler cannot deduce that?

Memory Hierarchy Optimizations: Data Layout

Data layout optimizations

- Array transpose to get stride-1 access
- Building cache-aware data structures by array merging
- Array padding
- Etc.

Data layout optimizations

- Stride-1 access is usually fastest for several reasons; particularly the reuse of cache line contents
- Data layout for multidimensional arrays in FORTRAN: column-major order

Example:
4x3 array
`A(i,j)`

Memory address

8	A(1,1)	A(2,1)	A(3,1)	A(4,1)
4	A(1,2)	A(2,2)	A(3,2)	A(4,2)
0	A(1,3)	A(2,3)	A(3,3)	A(4,3)

Data arrangement is „transpose“ of usual matrix layout

Data layout optimizations

Stride-1 access: innermost loop iterates over first index

- Either by choosing the right data layout (*array transpose*) or
- By arranging nested loops in the right order (*loop interchange*):

```
do i=1,N                          do j=1,M
  do j=1,M                          do i=1,N
    a(i,j)=a(i,j)+b(i,j)    ⟶        a(i,j)=a(i,j)+b(i,j)
  enddo                             enddo
enddo                             enddo
```

Stride-N access ⟶ **Stride-1 access**

This will usually be done by the compiler!

Data layout optimizations: Stride-1 access

```
do i=1,N
  do j=1,M
    s(i)=s(i)+b(i,j)*c(j)
  enddo
enddo
```

Better transpose matrix b so that inner loop gets stride 1

How about loop interchange in this case?

Data layout optimizations: Cache-aware data structures

- Idea: Merge data which are needed together to increase spatial locality: cache lines contain several data items
- Example: Gauss-Seidel iteration, determine data items needed simultaneously

$$u_i^{k+1} = a_{i,i}^{-1} \left(f_i - \sum_{j<i} a_{i,j}\, u_j^{k+1} - \sum_{j>i} a_{i,j}\, u_j^k \right)$$

Data layout optimizations: Cache-aware data structures

- Example (cont'd): right-hand side and coefficients are accessed simultaneously, reuse cache line contents by *array merging* => enhance spatial locality

```
typedef struct {
   double f;
   double c_N, c_E, c_S, c_W, c_C;
} equationData; // Data merged in memory

double u[N][N];                  // Solution vector
equationData rhsAndCoeff[N][N]; // Right-hand side
                                 // and coefficients
```


Data layout optimizations: Array padding

- Idea: Allocate arrays larger than necessary
  - Change relative memory distances
  - Avoid severe *cache thrashing* effects
- Example (FORTRAN: column-major order): Replace

```
double precision u(1024, 1024)
```

by

```
double precision u(1024+pad, 1024)
```

- How to choose `pad`?

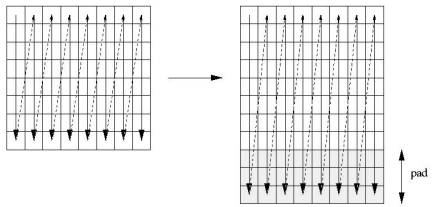
Data layout optimizations: Array padding

- C.-W. Tseng et al. (UMD):
  Research on cache modeling and compiler-based array padding:
  - *Intra-variable padding:* pad within arrays
  - ⇒ Avoid *self-interference* misses
  - *Inter-variable padding:* pad between different arrays
  - ⇒ Avoid *cross-interference* misses

Data layout optimizations: Array padding

- Padding in 2D; e.g., FORTRAN77:

```
double precision u(0:1024+pad,0:1024)
```



Memory Hierarchy Optimizations: Data Access

Loop optimizations

- Loop unrolling (see above)
- Loop interchange
- Loop fusion
- Loop split = loop fission = loop distribution
- Loop skewing
- Loop blocking
- Etc.

Data access optimizations: Loop fusion

- Idea: Transform successive loops into a single loop to enhance temporal locality
- Reduces cache misses and enhances cache reuse (exploit temporal locality)
- Often applicable when data sets are processed repeatedly (e.g., in the case of iterative methods)

Data access optimizations: Loop fusion (cont'd)

Before:

```
do i= 1,N
   a(i)= a(i)+b(i)
enddo
do i= 1,N
   a(i)= a(i)*c(i)
enddo
```

a is loaded into the cache twice (if sufficiently large)

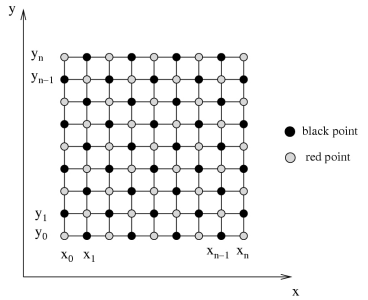
After:

```
do i= 1,N
   a(i)= (a(i)+b(i))*c(i)
enddo
```

a is loaded into the cache only once

Data access optimizations: Loop fusion (cont'd)

Example: red/black Gauss-Seidel iteration in 2D



Data access optimizations: Loop fusion (cont'd)

Code **before** applying loop fusion technique (standard implementation w/ efficient loop ordering, Fortran semantics: row major order):

```
for it= 1 to numIter do
   // Red nodes
   for i= 1 to n-1 do
      for j= 1+(i+1)%2 to n-1 by 2 do
         relax(u(j,i))
      end for
   end for
```

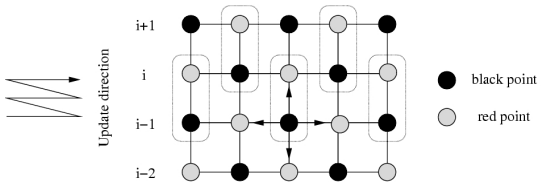
Data access optimizations: Loop fusion (cont'd)

```
   // Black nodes
   for i= 1 to n-1 do
      for j= 1+i%2 to n-1 by 2 do
         relax(u(j,i))
      end for
   end for
end for
```

This requires two sweeps through the whole data set per single GS iteration!

Data access optimizations: Loop fusion (cont'd)

How the fusion technique works:



Data access optimizations: Loop fusion (cont'd)

Code **after** applying loop fusion technique:

```
for it= 1 to numIter do
  // Update red nodes in first grid row
  for j= 1 to n-1 by 2 do
     relax(u(j,1))
  end for
```


Data access optimizations: Loop fusion (cont'd)

```
// Update red and black nodes in pairs
for i= 1 to n-1 do
   for j= 1+(i+1)%2 to n-1 by 2 do
        relax(u(j,i))
        relax(u(j,i-1))
   end for
end for
```

Data access optimizations: Loop fusion (cont'd)

```
// Update black nodes in last grid row
   for j= 2 to n-1 by 2 do
     relax(u(j,n-1))
   end for
```

Solution vector u passes through the cache only once instead of twice per GS iteration!

Data access optimizations: Loop split

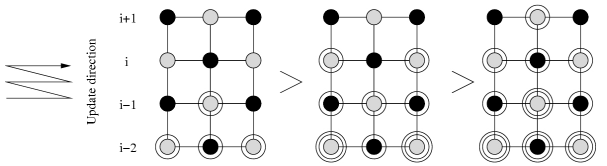
- The *inverse* transformation of loop fusion
- Divide work of one loop into two to make body less complicated
  - Leverage compiler optimizations
  - Enhance instruction cache utlization

Data access optimizations: Loop blocking

- *Loop blocking = loop tiling*
- Divide the data set into subsets (blocks) which are small enough to fit in cache
- Perform as much work as possible on the data in cache before moving to the next block
- This is not always easy to accomplish because of data dependencies

Data access optimizations: Loop blocking

Example: 1D blocking for red/black GS, respect the data dependencies!



Data access optimizations: Loop blocking

- Code **after** applying 1D blocking technique
- B = number of GS iterations to be blocked/combined

```
for it= 1 to numIter/B do
  // Special handling: rows 1, …, 2B-1
  // Not shown here …
```


Data access optimizations: Loop blocking

```
// Inner part of the 2D grid
for k= 2*B to n-1 do
  for i= k to k-2*B+1 by -2 do
    for j= 1+(k+1)%2 to n-1 by 2 do
      relax(u(j,i))
      relax(u(j,i-1))
    end for
  end for
end for
```

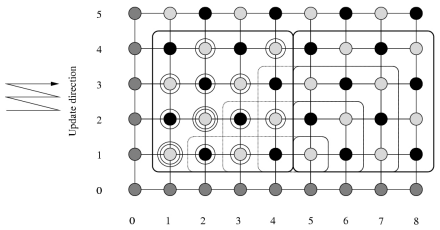

Data access optimizations: Loop blocking

```
  // Special handling: rows n-2B+1, …, n-1
  // Not shown here …
end for
```

- Result: Data is loaded once into the cache per B Gauss-Seidel iterations, if 2*B+2 grid rows fit in the cache simultaneously
- If grid rows are too large, 2D blocking can be applied

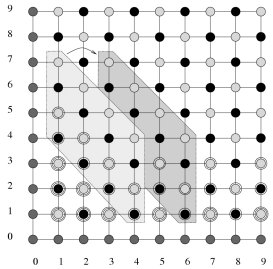
Data access optimizations Loop blocking

- More complicated blocking schemes exist
- Illustration: 2D square blocking



Data access optimizations: Loop blocking

- Illustration: 2D skewed blocking



Cache-optimized multigrid: DiMEPACK library

- DFG project DiME: Data-local iterative methods
- Fast algorithm + fast implementation
- Correction scheme: V-cycles, FMG
- Rectangular domains
- Constant 5-/9-point stencils
- Dirichlet/Neumann boundary conditions

DiMEPACK library

- C++ interface, fast Fortran77 subroutines
- Direct solution of the problems on the coarsest grid (LAPACK: LU, Cholesky)
- Single/double precision floating-point arithmetic
- Various array padding heuristics (Tseng)
- http://www10.informatik.uni-erlangen.de/dime

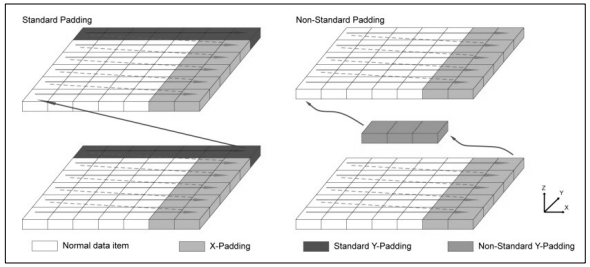
V(2,2) cycle - bottom line

Mflops	For what
13	Standard 5-pt. Operator
56	Cache optimized (loop orderings, data merging, simple blocking)
150	Constant coeff. + skewed blocking + padding
220	Eliminating rhs if 0 everywhere but boundary

Example: Cache-Optimized Multigrid on Regular Grids in 3D

Data layout optimizations for 3D multigrid

- Array padding



Data layout optimizations for 3D multigrid (cont'd)

- Standard padding in 3D; e.g., FORTRAN77:

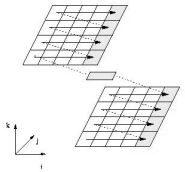
```
double precision u(0:1024,0:1024,0:1024)
```

becomes:

```
double precision u(0:1024+pad1,0:1024+pad2,0:1024)
```


Data layout optimizations for 3D multigrid (cont'd)

- Non-standard padding in 3D:

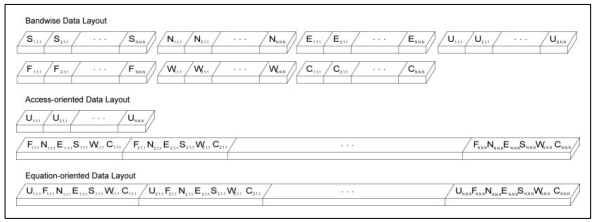


```
double precision u(0:1024+pad1,0:1024,0:1024)
      ...
u(i+k*pad2, j, k)
```

(or use hand-made index linearization – performance)?)

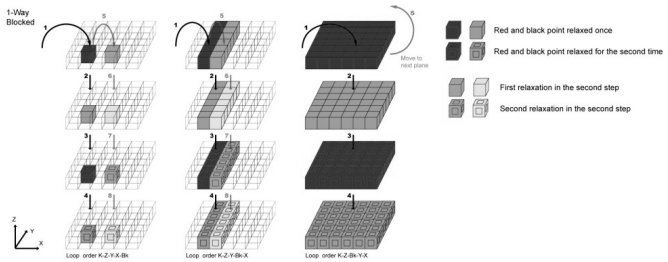
Data layout optimizations for 3D multigrid (cont'd)

- Array merging



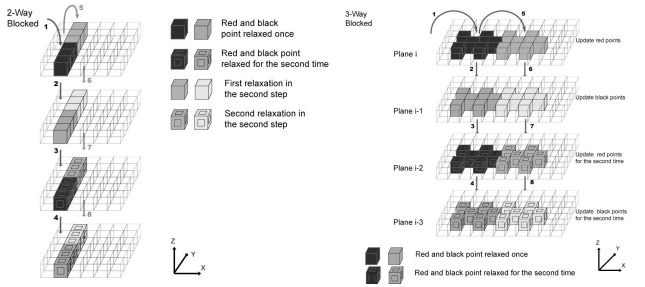
Data access optimizations for 3D multigrid (cont'd)

- 1-way blocking with loop-interchange



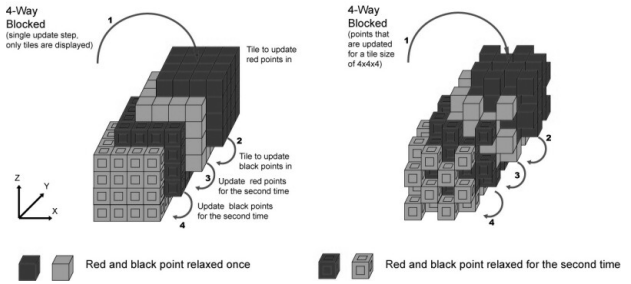
Data access optimizations for 3D multigrid (cont'd)

- 2-way blocking and 3-way blocking



Data access optimizations for 3D multigrid (cont'd)

- 4-way blocking



Example: Cache Optimizations for the Lattice Boltzmann Method

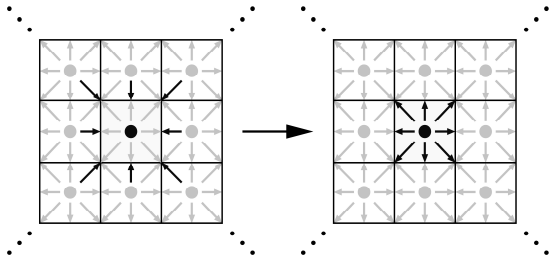
Lattice Boltzmann method

- Mainly used in CFD applications
- Employs a regular grid structure (2D, 3D)
- Particle-oriented approach based on a microscopic model of the moving fluid particles
- Jacobi-like cell update pattern: a single *time step* of the LBM consits of
  - *stream step and*
  - *collide step*

Demo

LBM

- Stream: read *distribution functions* from neighbors
- Collide: re-compute own distribution functions

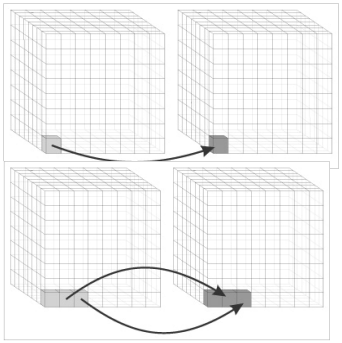


Data layout optimization

Layout 1:

Two separate grids

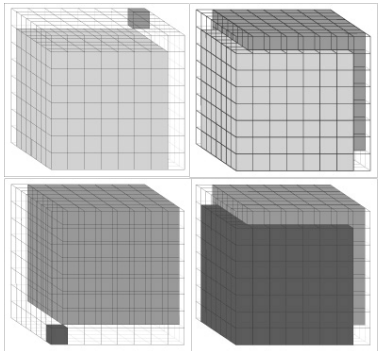
(standard approach)



Data layout optimization (cont'd)

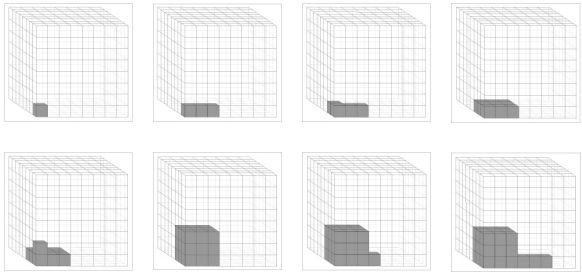
Layout 2:

Grid Compression:
save memory,
enhance locality



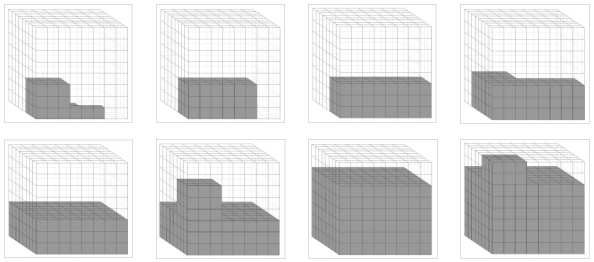
Data access optimizations

Access pattern 1: 3-way blocking:



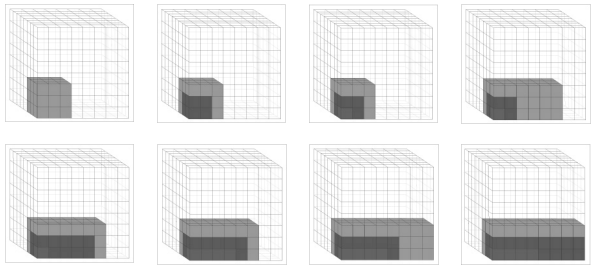
Data access optimizations (cont'd)

3-way blocking (cont'd):



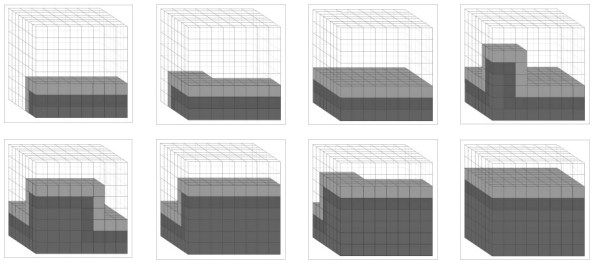
Data access optimizations (cont'd)

Access pattern 2: 4-way blocking:



Data access optimizations (cont'd)

4-way blocking (cont'd):



Data access optimizations (cont'd)

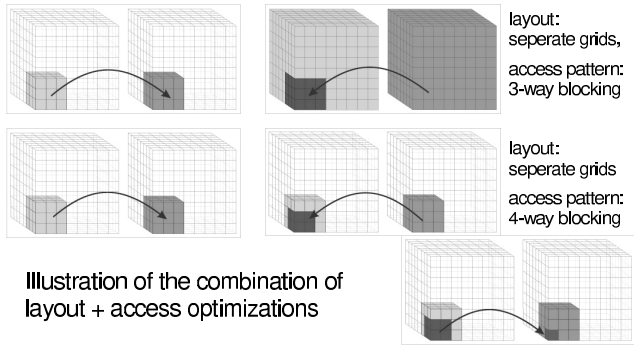
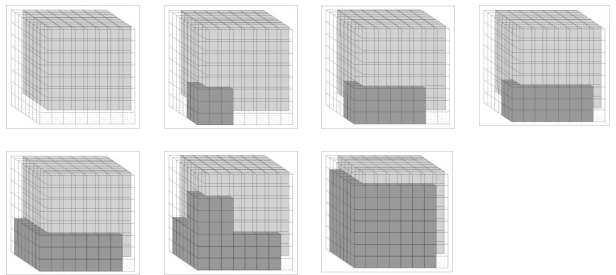


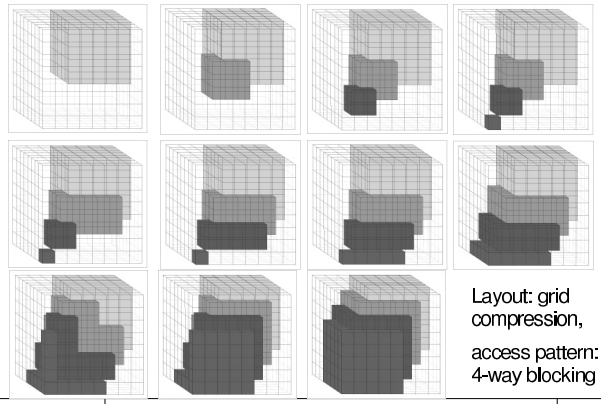
Illustration of the combination of layout + access optimizations

Data access optimizations (cont'd)

Layout: Grid compression, access pattern: 3-way blocking:



Data access optimizations (cont'd)

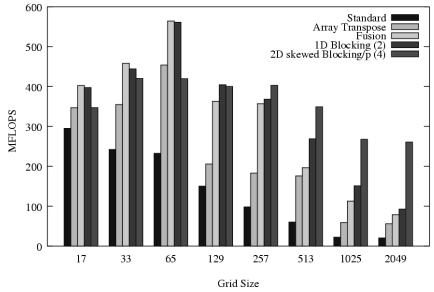


Layout: grid compression,

access pattern: 4-way blocking

Performance results

MFLOPS for 2D GS, const. coeff.s, 5-pt.,
DEC PWS 500au, Alpha 21164 CPU, 500 MHz



Memory access behavior

- Digital PWS 500au, Alpha 21164 CPU
- L1 = 8 KB, L2 = 96 KB, L3 = 4 MB
- We use DCPI to obtain the performance data
- We measure the percentage of accesses which are satisfied by each individual level of the memory hierarchy
- Comparison: standard implementation of red/black GS (efficient loop ordering) vs. 2D skewed blocking (with and without padding)

Memory access behavior (cont'd)

- Standard implementation of red/black GS, without array padding

Size	+/-	L1	L2	L3	Mem.
33	4.5	63.6	32.0	0.0	0.0
65	0.5	75.7	23.6	0.2	0.0
129	-0.2	76.1	9.3	14.8	0.0
257	5.3	55.1	25.0	14.5	0.0
513	3.9	37.7	45.2	12.4	0.8
1025	5.1	27.8	50.0	9.9	7.2
2049	4.5	30.3	45.0	13.0	7.2

Memory access behavior (cont'd)

- 2D skewed blocking without array padding, 4 iterations blocked (B = 4)

Size	+/-	L1	L2	L3	Mem.
33	27.4	43.4	29.1	0.1	0.0
65	33.4	46.3	19.5	0.9	0.0
129	36.9	42.3	19.1	1.7	0.0
257	38.1	34.1	25.1	2.7	0.0
513	38.0	28.3	27.0	6.7	0.1
1025	36.9	24.9	19.7	17.6	0.9
2049	36.2	25.5	0.4	36.9	0.9

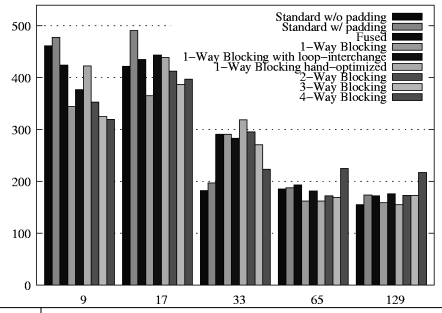
Memory access behavior (cont'd)

- 2D skewed blocking with appropriate array padding, 4 iterations blocked (B = 4)

Size	+/-	L1	L2	L3	Mem.
33	28.2	66.4	5.3	0.0	0.0
65	34.3	55.7	9.1	0.9	0.0
129	37.5	51.7	9.0	1.9	0.0
257	37.8	52.8	7.0	2.3	0.0
513	38.4	52.7	6.2	2.4	0.3
1025	36.7	54.3	6.1	2.0	0.9
2049	35.9	55.2	6.0	1.9	0.9

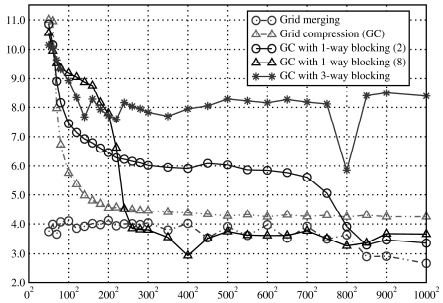
Performance results

3D MG, F77, var. coeff.s, 7-pt.,
Intel Pentium4, 2.4 GHz, Intel ifc V7.0 compiler



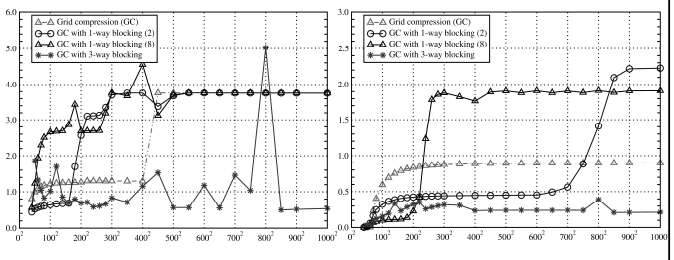
Performance results (cont'd)

2D LBM (D2Q9), C(++), AMD Athlon XP 2400+, 2.0 GHz, Linux, gcc V3.2.1 compiler



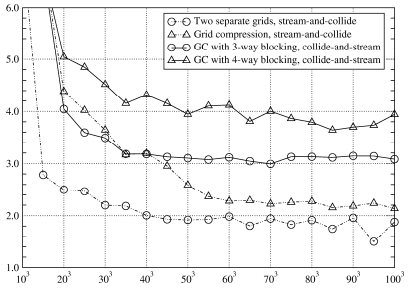
Performance results (cont'd)

Cache behavior (left: L1, right: L2) for previous experiment, measured with PAPI



Performance results (cont'd)

3D LBM (D3Q19), C, AMD Opteron, 1.6 GHz, Linux, gcc V3.2.2 compiler



Optimizations for Computations on Unstructured Grids

See C.C. Douglas et.al.
http://www.ccs.uky.edu/~douglas/ccd-kfcs.html

References

- **S. Goedecker, A. Hoisie: Performance Optimization of Numerically Intensive Codes, SIAM, 2001.**
- C.C. Douglas, G. Haase, J. Hu, W. Karl, M. Kowarschik, U. Rüde, C. Weiss, Portable memory hierarchy techniques for PDE solvers, Part I, SIAM News 33/5 (2000), pp. 1, 8-9. Part II, SIAM News 33/6 (2000), pp. 1, 10-11, 16.
- C. C. Douglas, J. Hu, W. Karl, M. Kowarschik, U. Rüde, C. Weiss, Cache optimization for structured and unstructured grid multigrid, Electronic Transactions on Numerical Analysis, 10 (2000), pp. 25-40.
- J. Handy, The Cache Memory Book, 2nd ed., Academic Press, 1998
- J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, 2nd ed., Morgan Kauffmann Publishers, 1996.

References (cont'd)

- M. Kowarschik, C. Weiss, DiMEPACK – A Cache-Optimized Multigrid Library, Proc. of the Intl. Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2001), vol. I, June 2001, pp. 425-430.
- U. Rüde, Iterative Algorithms on High Performance Architectures, Proc. of the EuroPar97 Conference, Lecture Notes in Computer Science, Springer, Berlin, 1997, pp. 57-71.
- U. Rüde, Technological Trends and their Impact on the Future of Supercomputing, H.-J. Bungartz, F. Durst, C. Zenger (eds.), High Performance Scientific and Engineering Computing, Lecture Notes in Computer Science, Vol. 8, Springer, 1998, pp. 459-471.
- D. Bulka, D. Mayhew, Efficient C++, Addison-Wesley, 2000

References (cont'd)

- U. Trottenberg, A. Schuller, C. Oosterlee, Multigrid, Academic Press, 2000.
- C. Weiss, W. Karl, M. Kowarschik, U. Rüde, Memory Characteristics of Iterative Methods. In Proceedings of the Supercomputing Conference, Portland, Oregon, November 1999.
- C. Weiss, M. Kowarschik, U. Rüde, and W. Karl, Cache-aware Multigrid Methods for Solving Poisson's Equation in Two Dimension. Computing, 64(2000), pp. 381-399.

Related websites

- http://www10.informatik.uni-erlangen.de/dime
- http://www.ccs.uky.edu/~douglas/ccd-kfcs.html
- http://www.mgnet.org
- http://www.fz-juelich.de/zam/PCL
- http://icl.cs.utk.edu/projects/papi
- http://www.tru64unix.compaq.com/dcpi