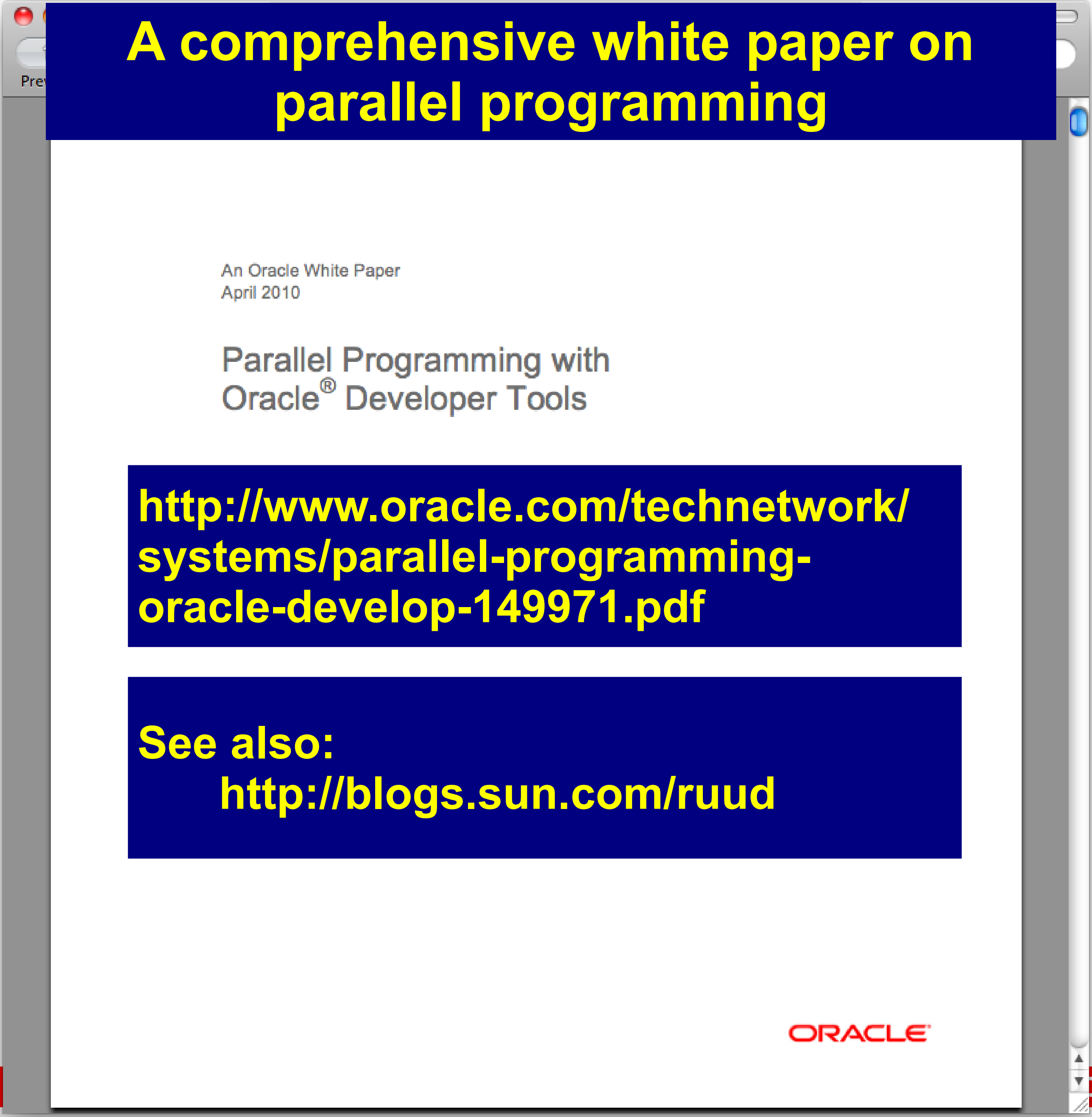
ORACLE®

Performance Tuning Techniques for Cache Based Systems

Ruud van der Pas
Senior Staff Engineer - Architecture and Performance
SPARC Microelectronics, Oracle, Santa Clara, CA, USA

A comprehensive white paper on parallel programming



An Oracle White Paper
April 2010

Parallel Programming with
Oracle® Developer Tools

http://www.oracle.com/technetwork/systems/parallel-programming-oracle-develop-149971.pdf

See also:
http://blogs.sun.com/ruud

Outline

- Motivation
- The Memory Hierarchy
- Oracle Solaris Studio
- Loop Based Optimizations
- Instruction Scheduling Optimizations
- Performance Considerations

Motivation

Why This Topic At A Seminar Called

“Parallel Programming for Computational Engineering & Science” ?

Because Serial Performance Matters

A Lot .....

Why Does It Matter?

It Makes The Program Go Faster

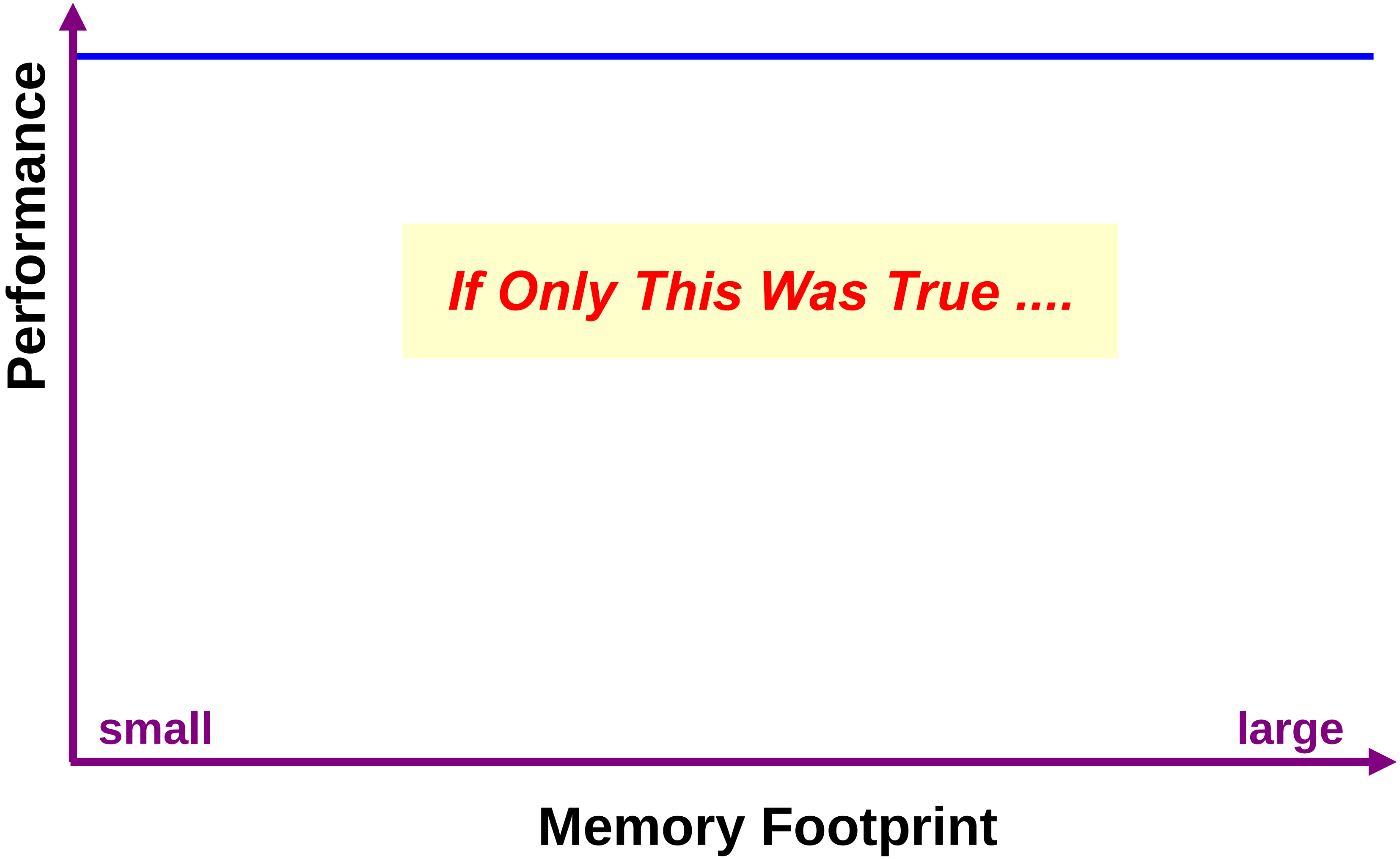
Of Great Benefit To Scalability

Amdahl's Law and Bandwidth

It Is Fun To Do !

The Memory Hierarchy

Intuitive Performance Graph

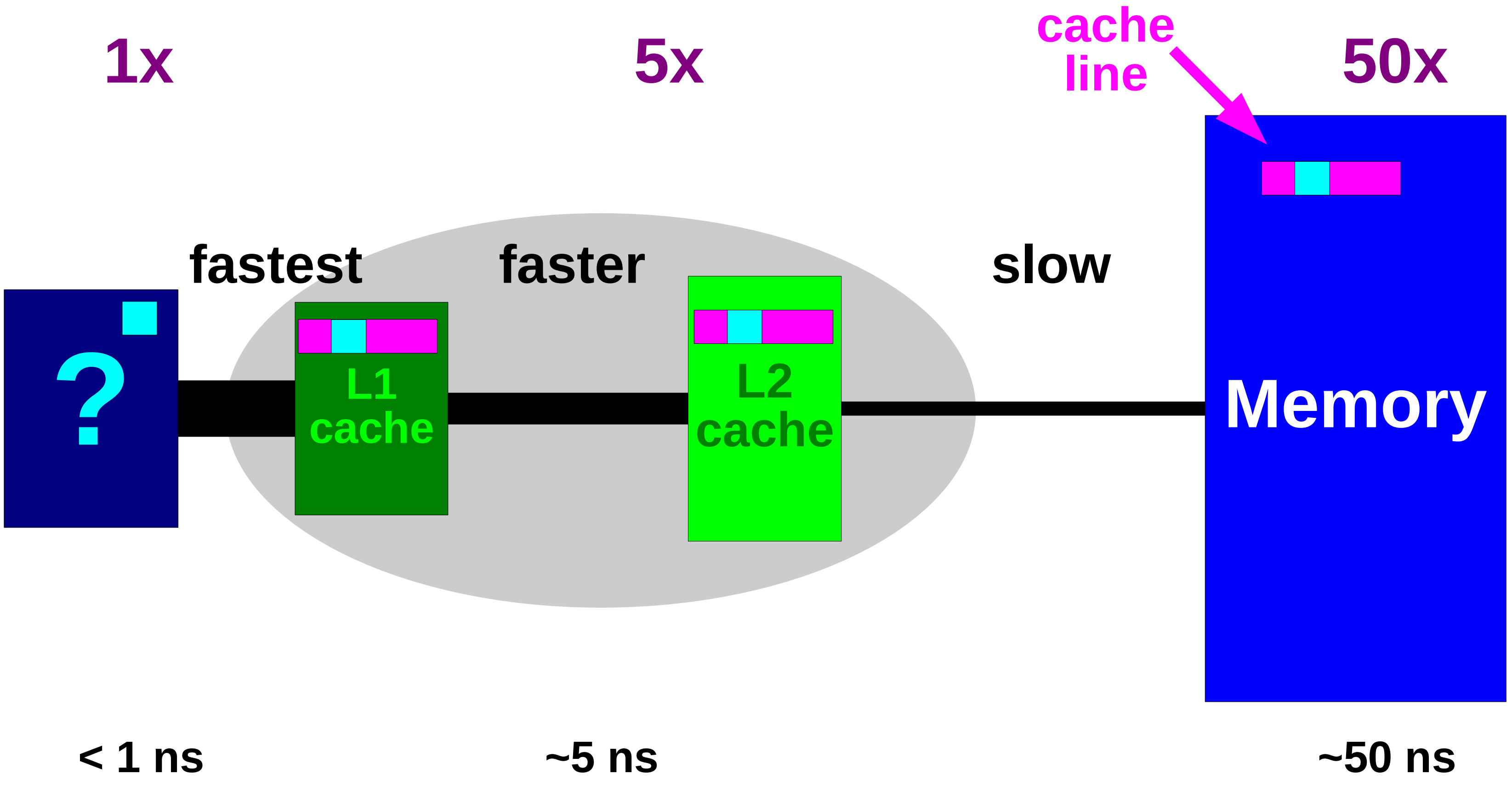


If Only This Was True ....

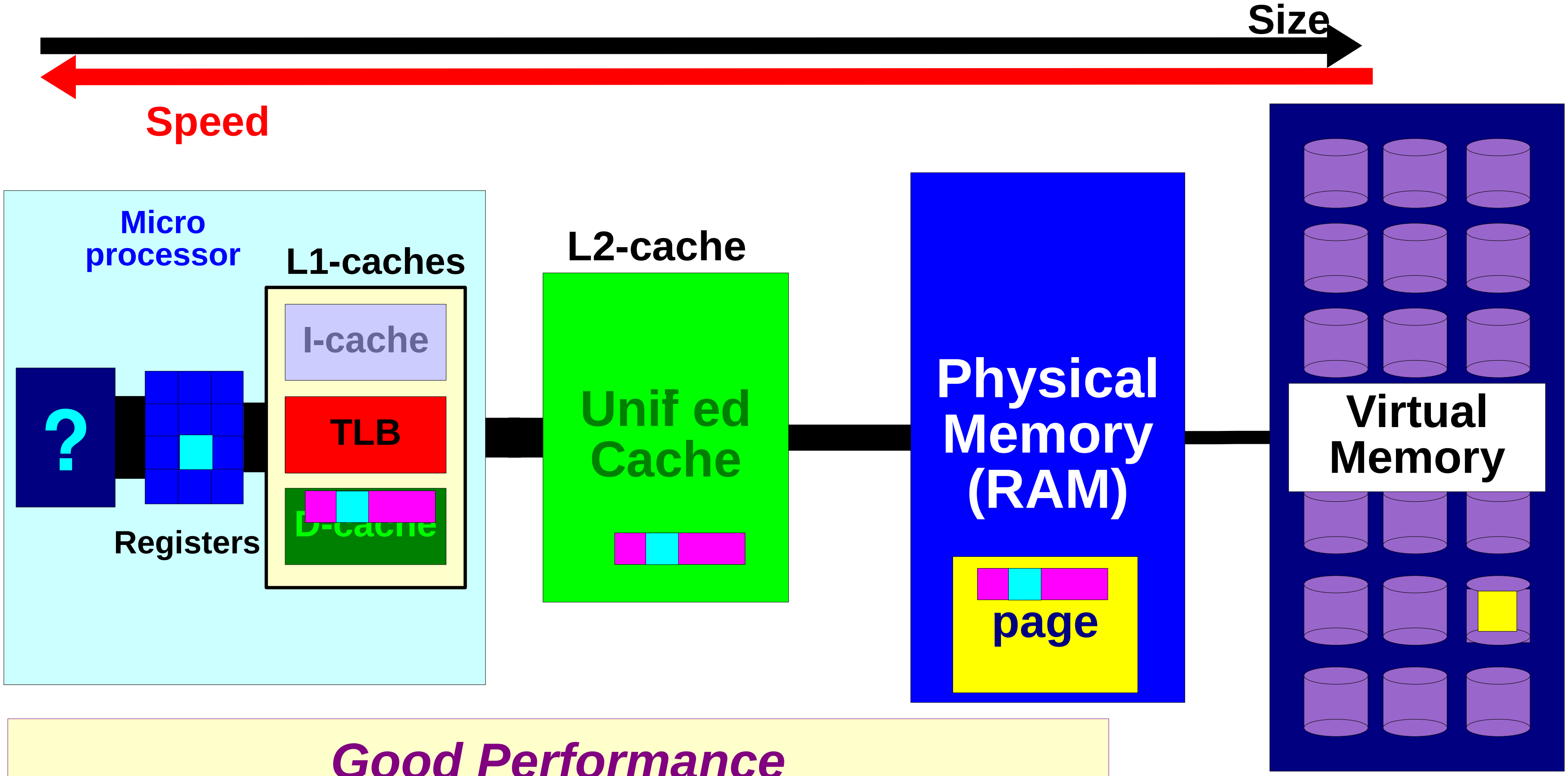
About Memory

- Memory plays a crucial role in performance
- Not accessing memory in the right way degrades performance on all computer systems
- The extent of the degradation depends on the system
- Knowing more about some of the relevant memory characteristics helps you to write code such that the problem is non-existent, or at least minimal

A Typical Cache Based System



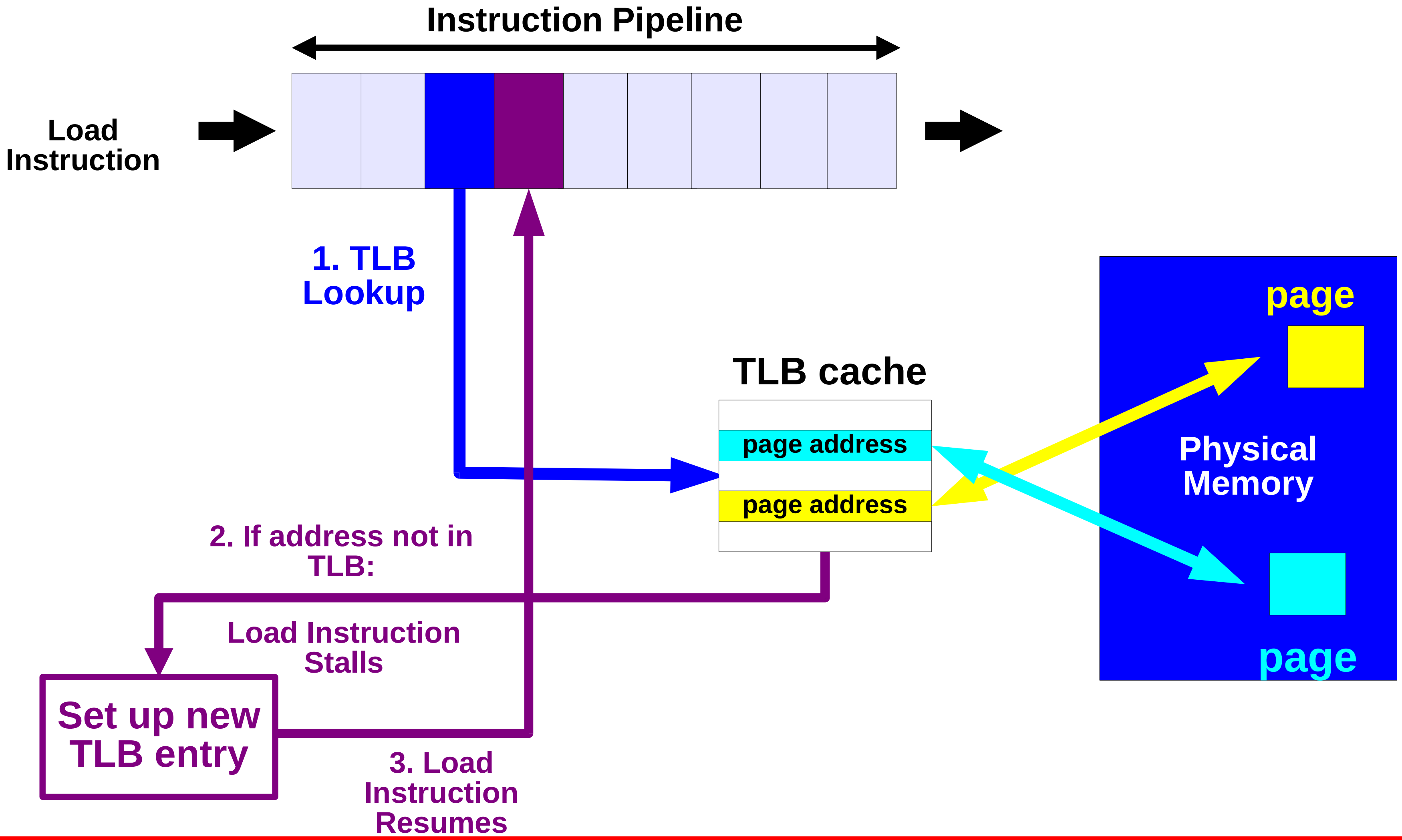
The Memory Hierarchy



__Good Performance__

Get Data Into CPU As Fast As Possible And Keep It There For As Long As You Can

The TLB ('Page Address Cache')



Instruction Pipeline

Load Instruction

1. TLB Lookup

TLB cache

page address

page address

Physical Memory

page

page

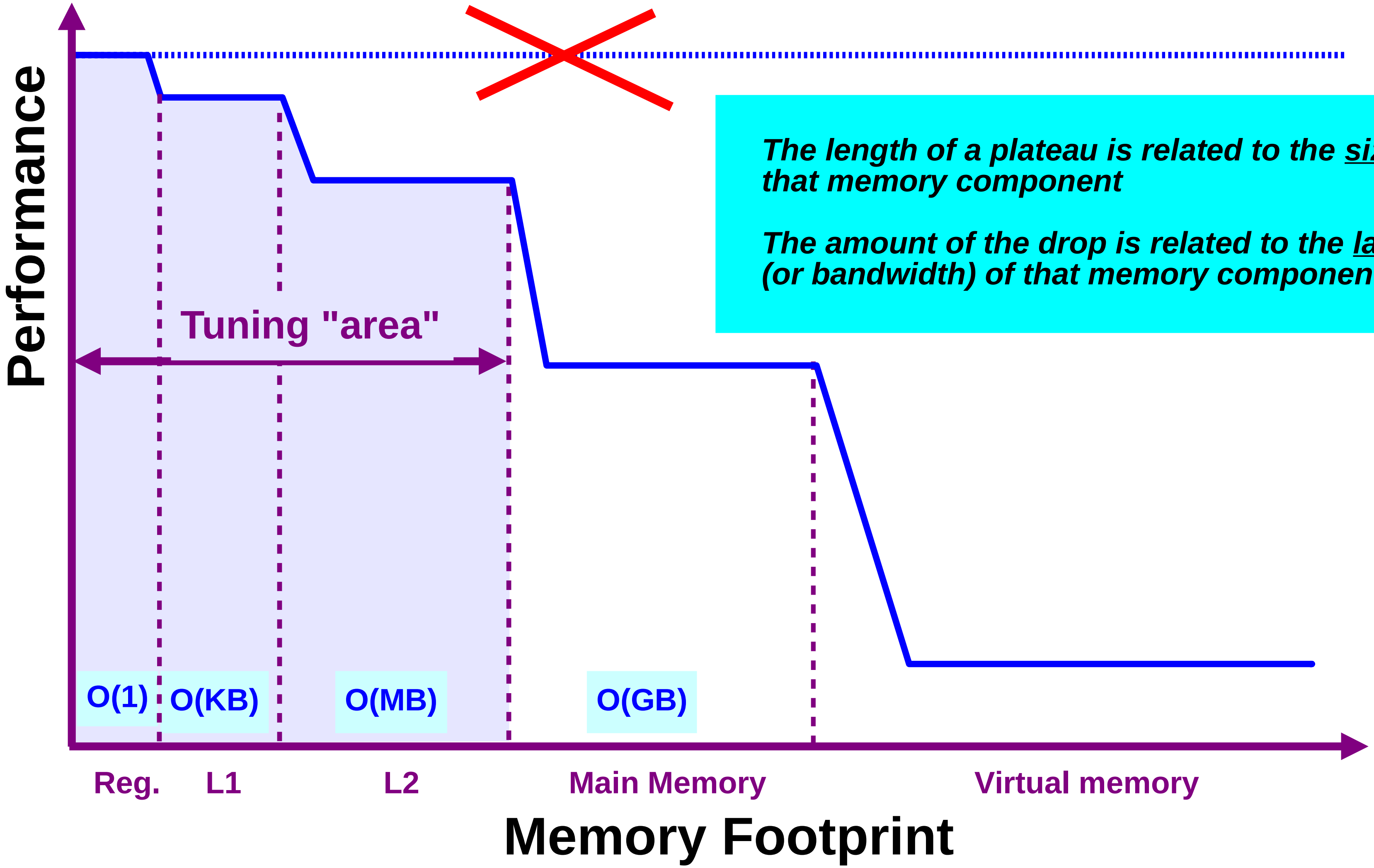
2. If address not in TLB:

Load Instruction Stalls

Set up new TLB entry

3. Load Instruction Resumes

Performance Is Not Uniform

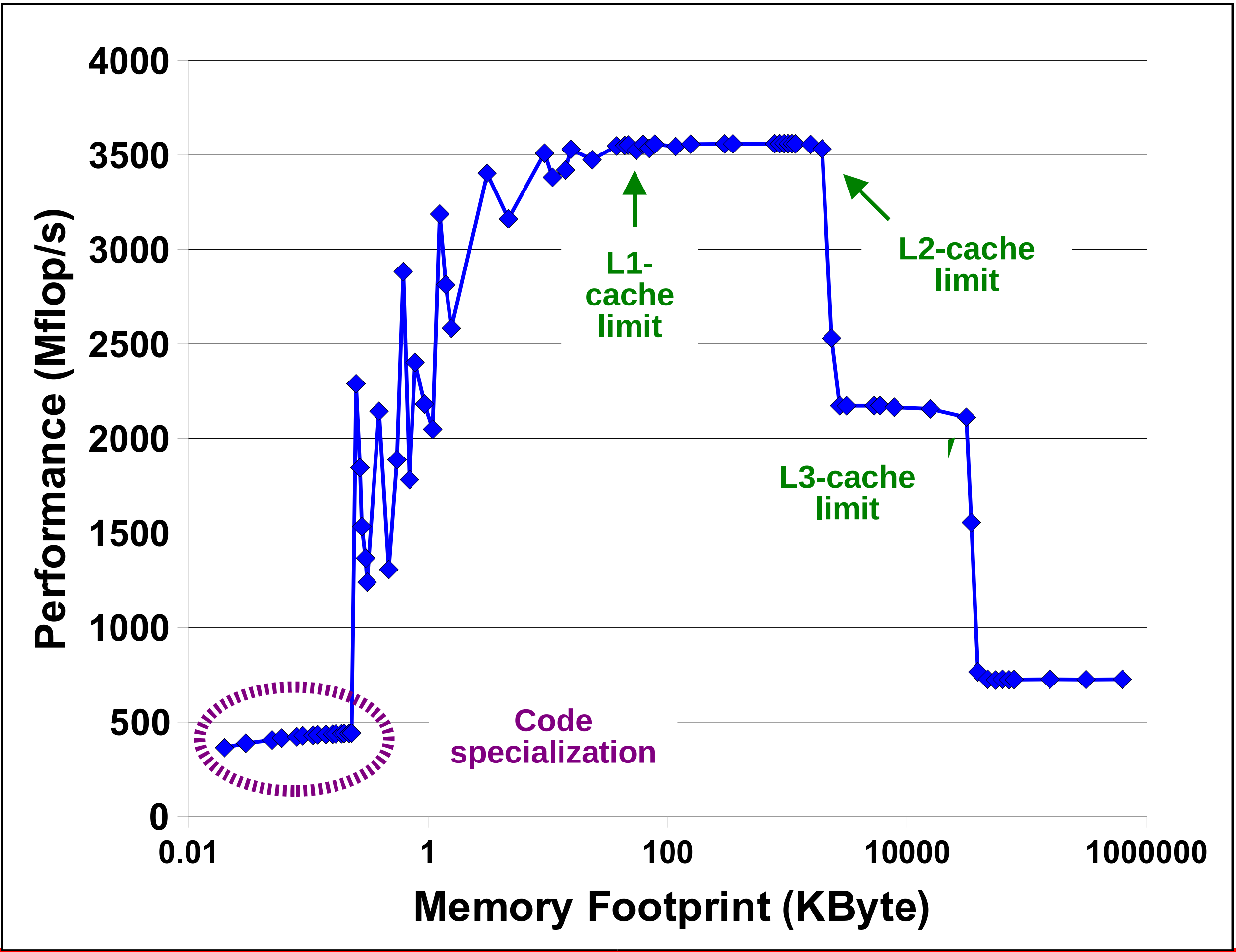


The length of a plateau is related to the size of that memory component

The amount of the drop is related to the latency (or bandwidth) of that memory component

Example - 13th deg. polynomial

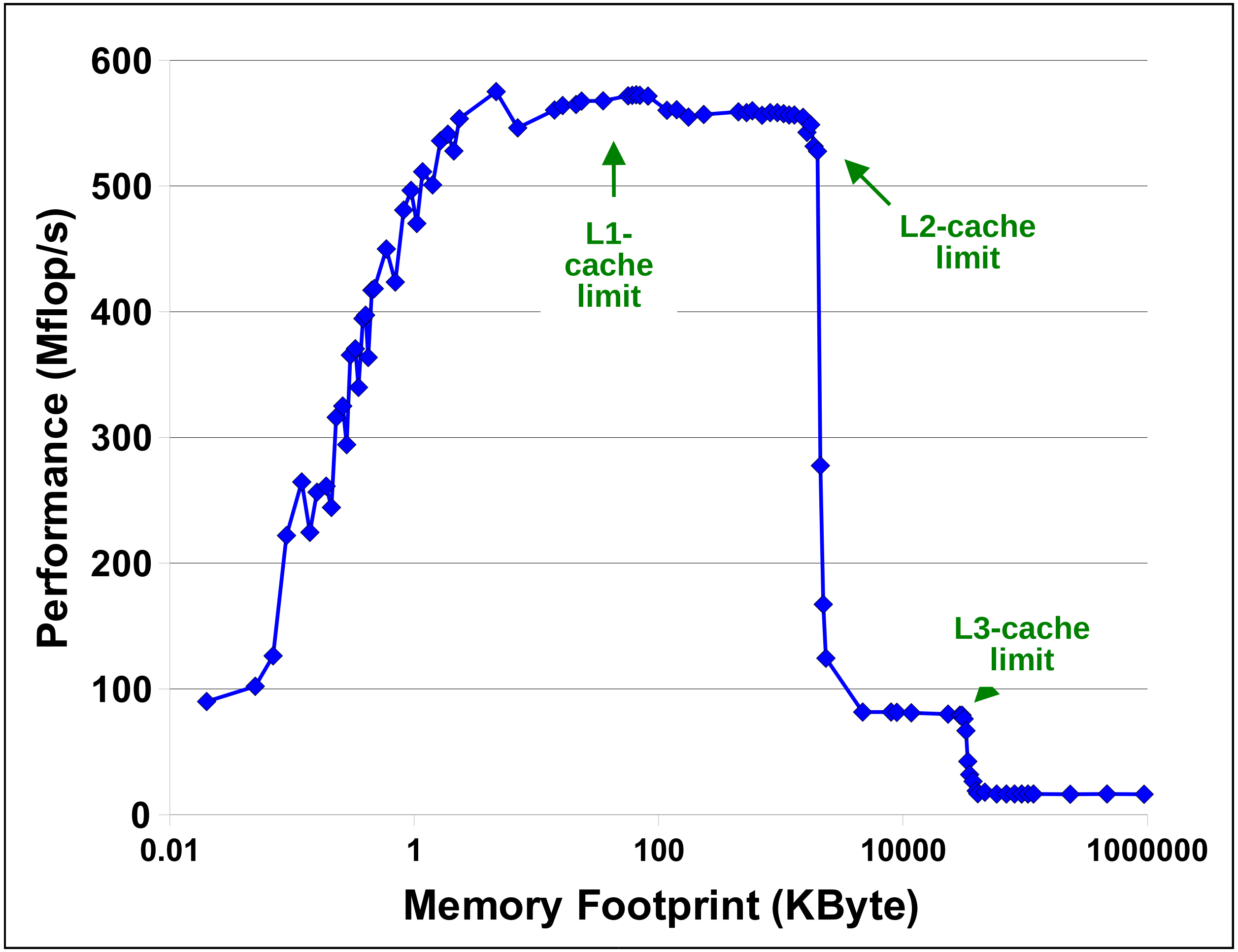
```
for (i=0; i<vlen; i++)
   p[i] = c[0] + q[i]*(c[1] + q[i]*(c[2] + q[i]*(c[3] + ....
```



- ***This operation is <u>CPU bound</u> i.e. there are many more floating point operations than memory references***
- ***The system realizes 99% of the absolute peak performance !***
- ***Note the start-up effect and the performance drop for larger problems***

Example - Vector Addition

```
for (i=0; i<vlen; i++)
   p[i] = q[i] + r[i];
```

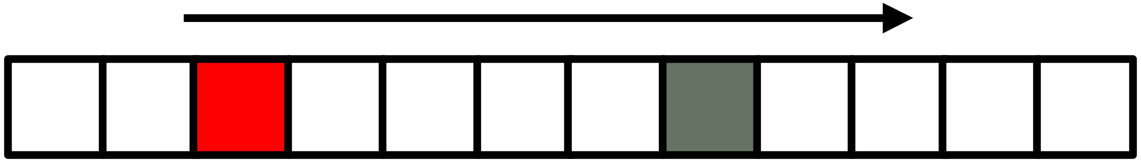


- ***This operation is <u>memory bound</u> i.e. there are more memory references than f oating point operations***
- ***The system realizes close to the theoretical peak performance for <u>this</u> operation (i.e. 1/6 of absolute peak)***
- ***Note the start-up effect and the performance drop for larger problems***

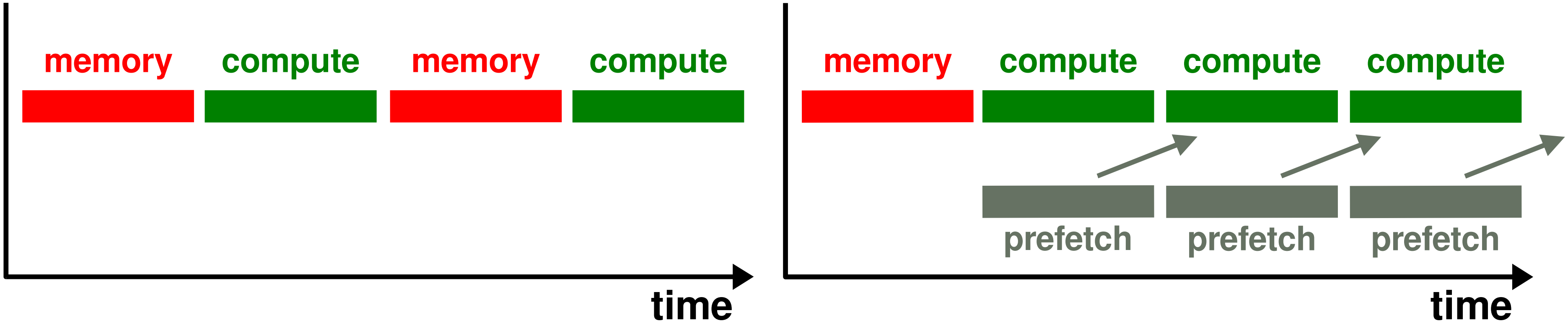
Hiding Memory Latency

- ***The memory access pattern may be predictable:***

  Example: summation of elements

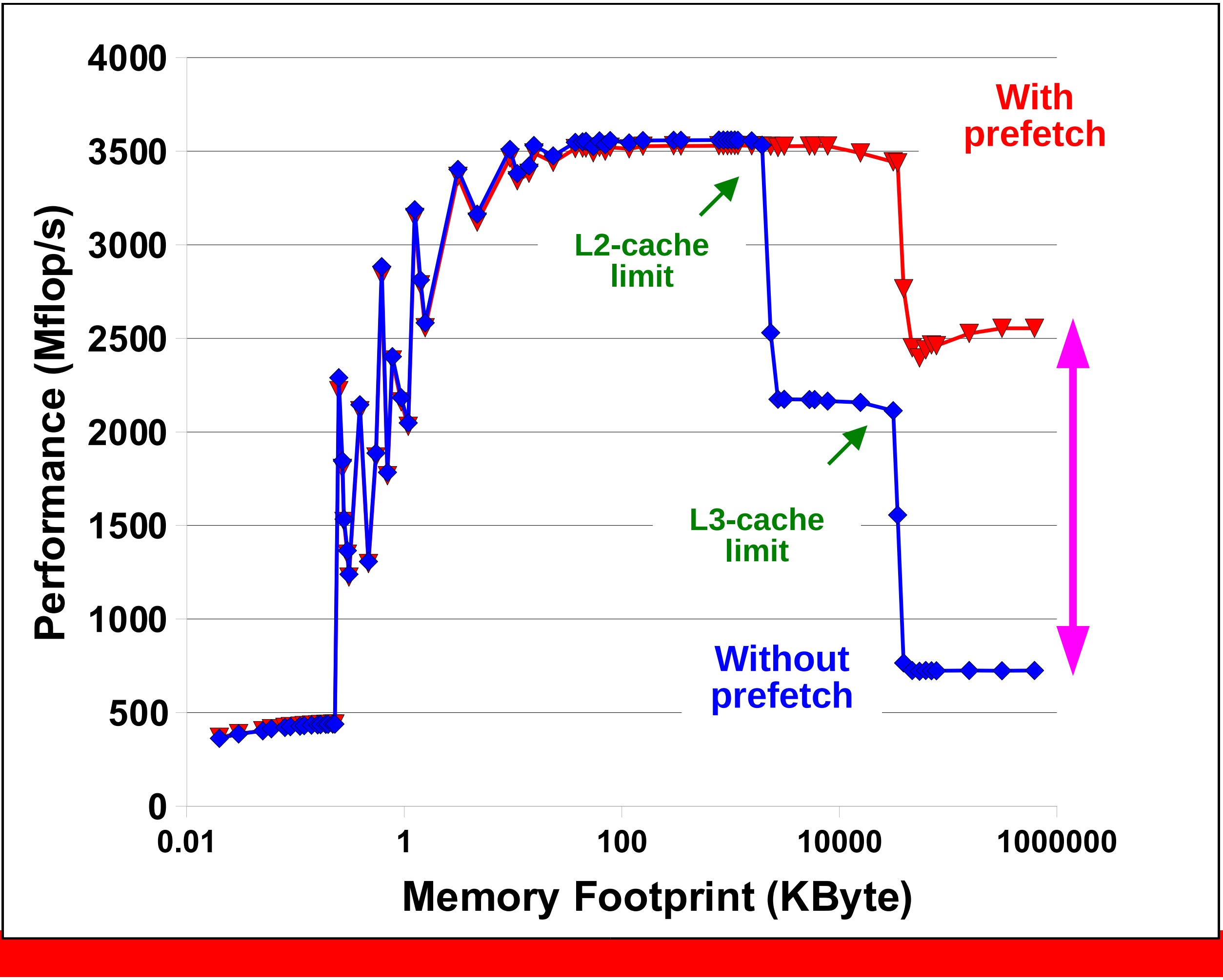


- ***With prefetch, one fetches memory before it is needed***
- ***This is called a "latency hiding technique"***



Prefetch - 13th deg. polynomial

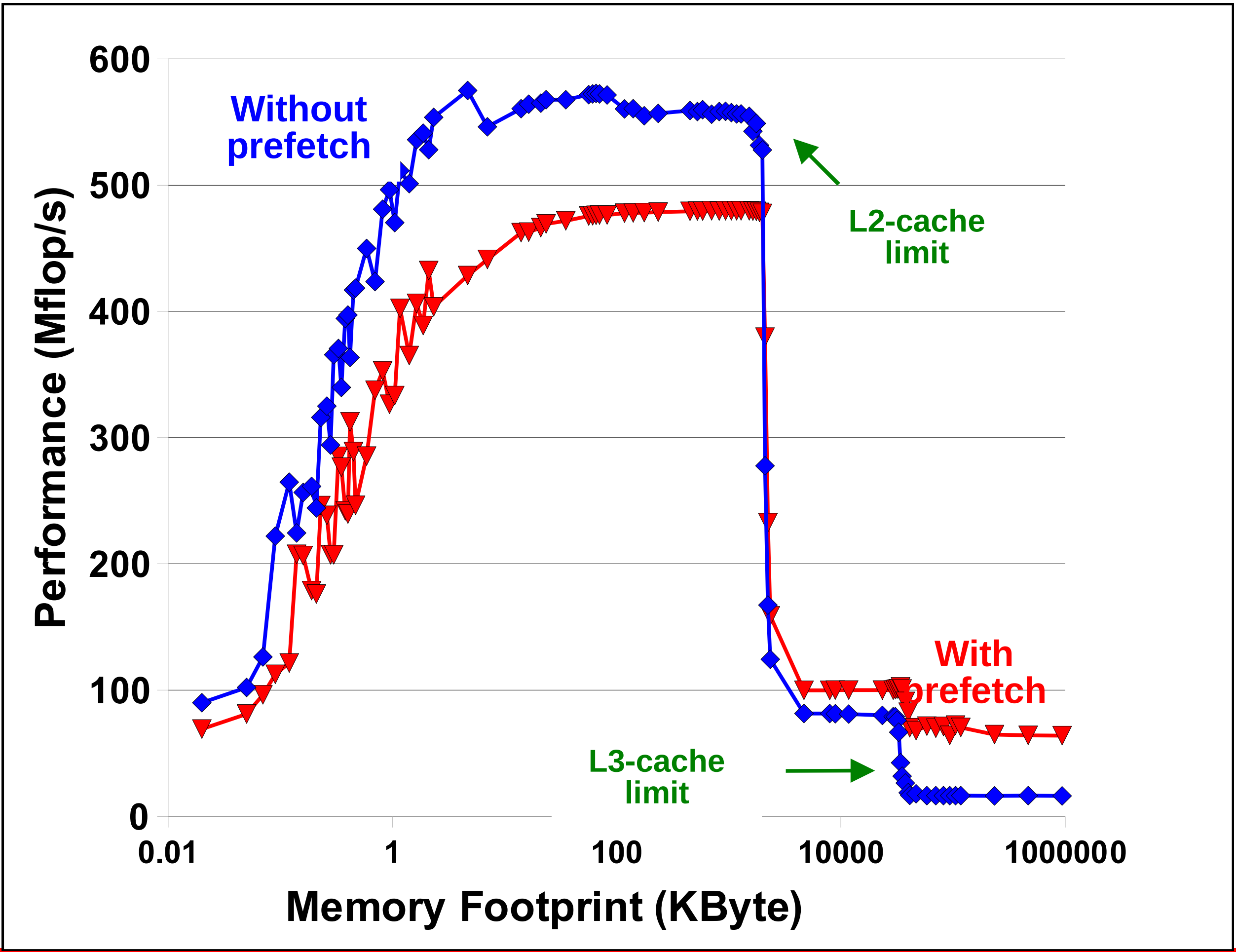
```
for (i=0; i<vlen; i++)
   p[i] = c[0] + q[i]*(c[1] + q[i]*(c[2] + q[i]*(c[3] + ....
```



- *Re-compiled for automatic prefetch*
- *Performance for L2 resident problem sizes is the same*
- *Hides latency to L3 cache !*
- *For large problem sizes, automatic prefetch is a big win !*

Prefetch - Vector Addition

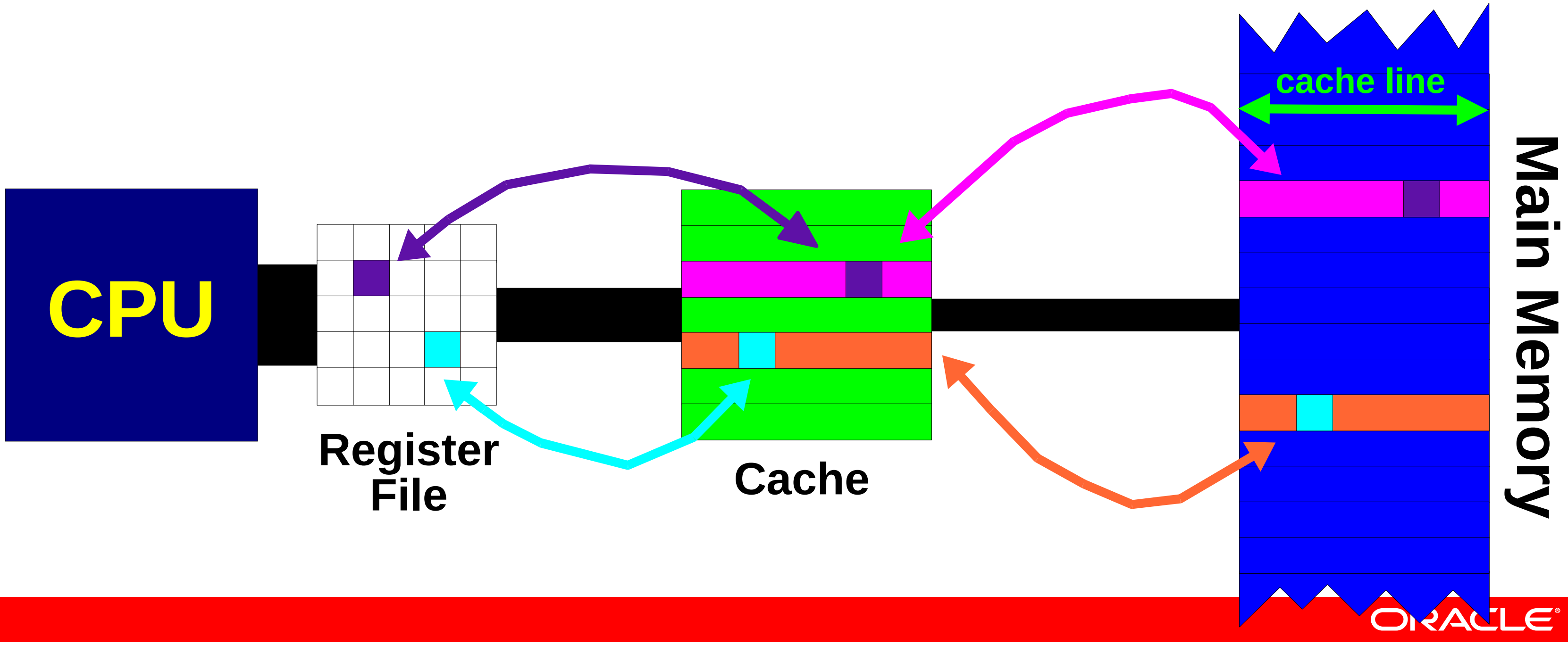
```
for (i=0; i<vlen; i++)
   p[i] = q[i] + r[i];
```



- *Re-compiled for automatic prefetch*
- *Performance for L2 resident problem sizes is a little less if prefetch is used*
- *For large problem sizes, automatic prefetch gives a significant performance improvement*

Cache Lines

- ❑ *For good performance, it is crucial to use the cache(s) in the intended (=optimal) way*
- ❑ *Recall that the unit of transfer is a cache "line"*
- ❑ *A cache line is a linear structure i.e. it has a fixed length (in bytes) and a starting address in memory*



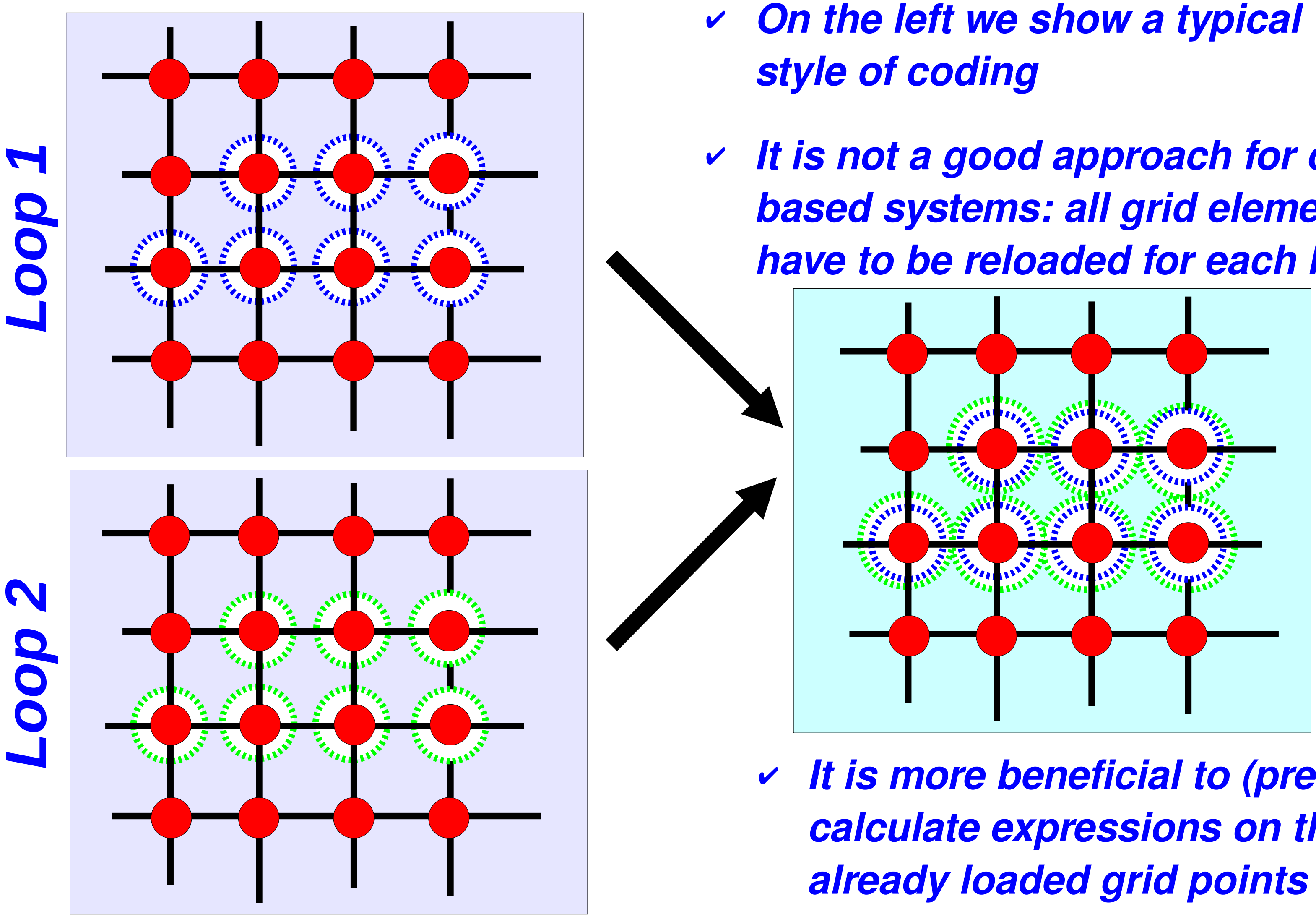
Cache Line Utilization

Two Key Rules - Maximize

- *Spatial Locality - Use all data in one cache line*
  - ✓ *This strongly depends on the storage of your data and the access pattern(s)*
- *Temporal Locality - Re-use data in a cache line*
  - ✓ *This mainly depends on the algorithm used*

Cache Line Re-use

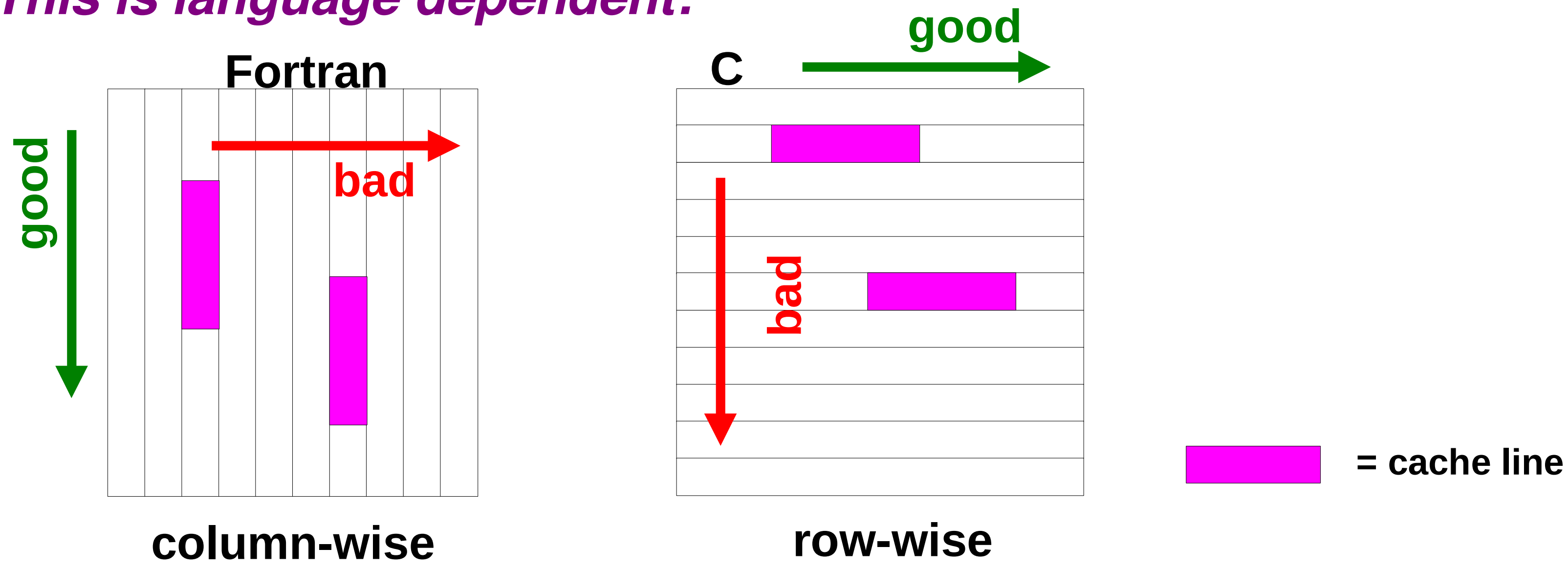
- ✔ *On the left we show a typical 'vector' style of coding*
- ✔ *It is not a good approach for cache based systems: all grid elements have to be reloaded for each loop*



- ✔ *It is more beneficial to (pre-) calculate expressions on the already loaded grid points*

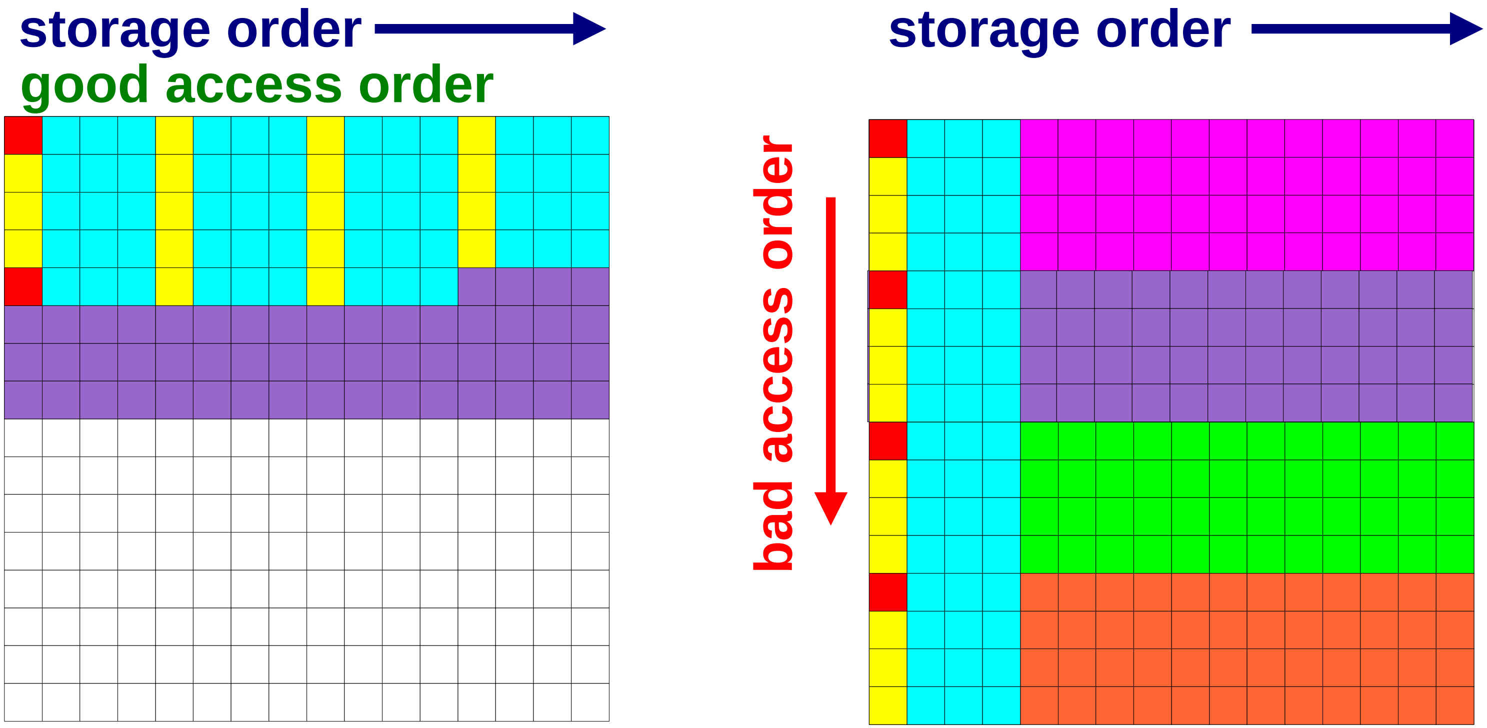
Memory Access

- **Memory has a 1D, linear, structure**
  - *Access to multi-dimensional arrays depends on the way data is stored*
  - *This is language dependent:*



Bad Memory Access Has A Huge Impact On Performance

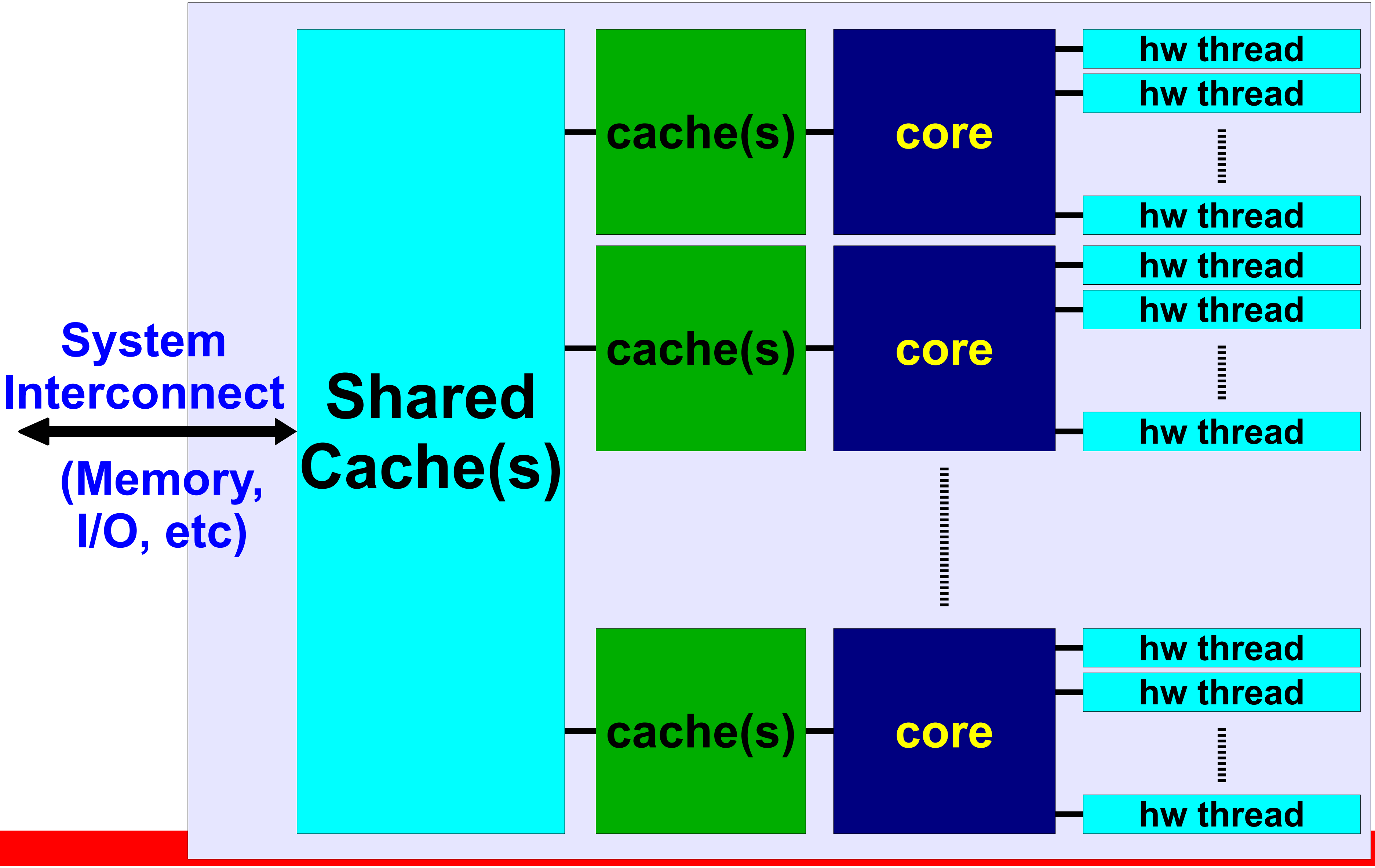
Bad Memory Access (C Example)



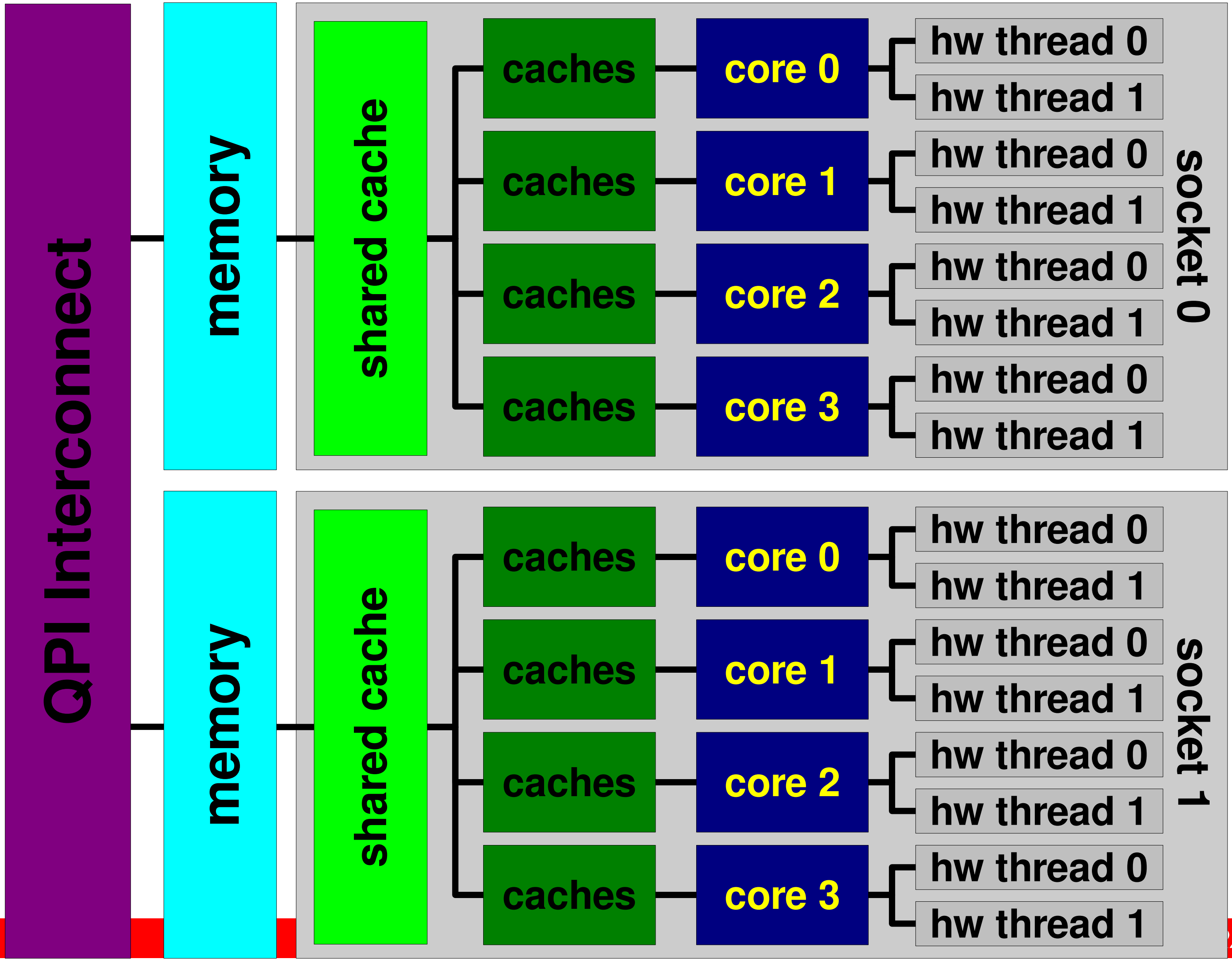
- 🟥 = TLB miss
- 🟨 = D-cache miss
- 🟦 = Cached elements
- 🟪 🟪 = Virtual memory page

- ✔ *If the entire matrix fits in the cache, the access pattern hardly matters*
- ✔ *For out-of-cache matrices however, the access pattern <u>does</u> matter*
- ✔ *With a bad memory access pattern, we get many more D-cache and TLB misses*

A Generic Multicore Architecture



A Two Socket Nehalem System



The Memory Hierarchy

Type	Level	Sharing level	Capacity
L1	Data	core	32 KB
	D-TLB	core	64 @ 4K
			32 @ 2M/4M
	Instructions	core	32 KB
	I-TLB	core	128 @ 4K
			7 @ 2M/4M
L2	Unified (Data and Instructions)	core	256 KB
	Unified TLB	core (?)	512 @ 4K
L3	Unified	4 cores	8 MB

System Characteristics

- ❑ *A two socket Nehalem ("Xeon X5570") system*
- ❑ *Clock speed of 2.93 GHz*
- ❑ *Operating System:*
  - *CentOS*
  - *Linux kernel 2.6.18*
- ❑ *Compiler: Oracle Solaris Studio 12 Update 1*

The Experiment

```
double compute_sum(int m, int n, double *a)
{
   double sum = 0.0;;

   for (int i=0; i<m; i++)
       for (int j=0; j<n; j++)
           sum += a[i][j];

   return(sum);
}
```

Good Memory Access

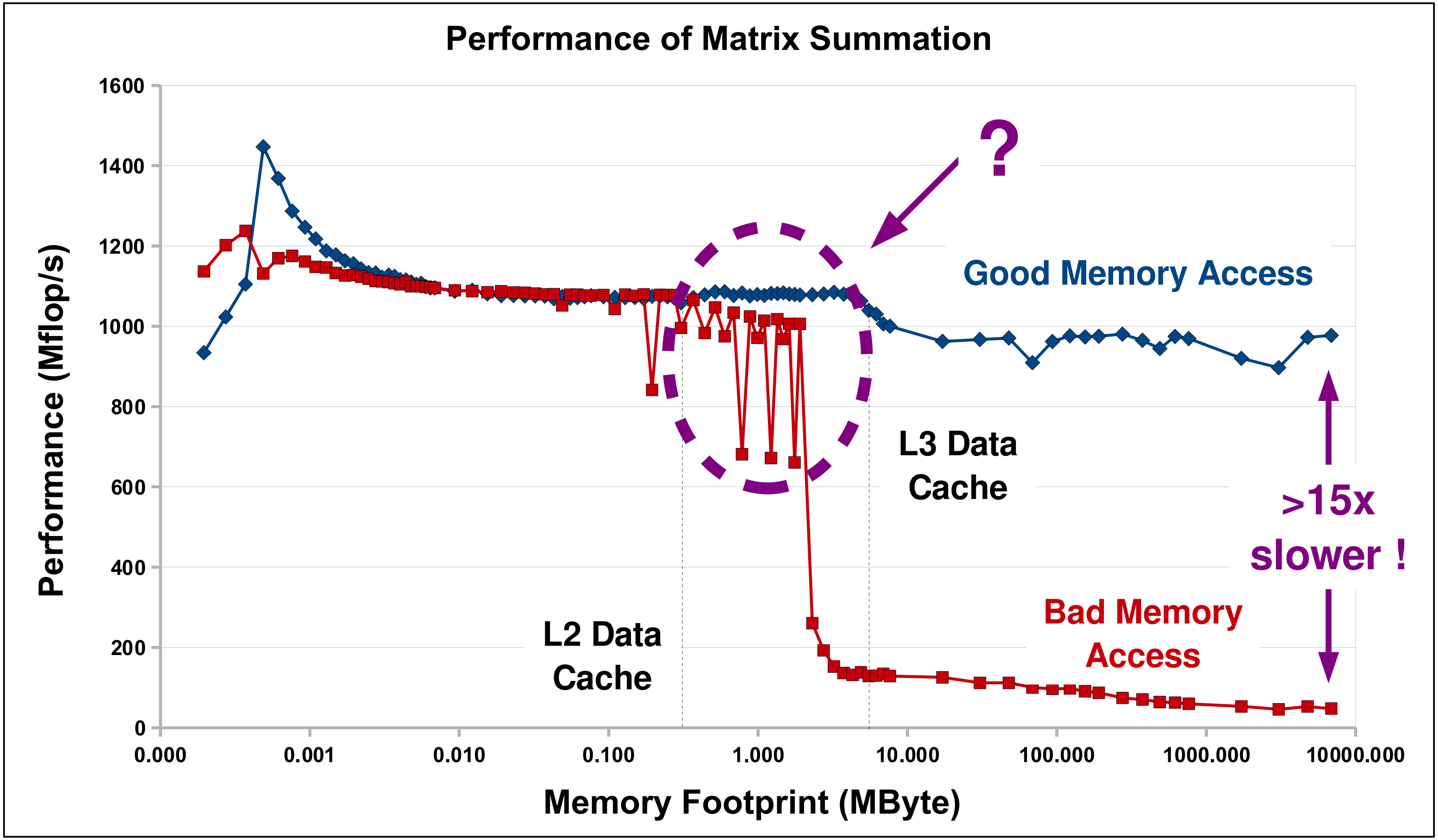
```
double compute_sum(int m, int n, double *a)
{
   double sum = 0.0;;

   for (int j=0; j<n; j++)
      for (int i=0; i<m; i++)
         sum += a[i][j];

   return(sum);
}
```

Bad Memory Access

Performance on Nehalem

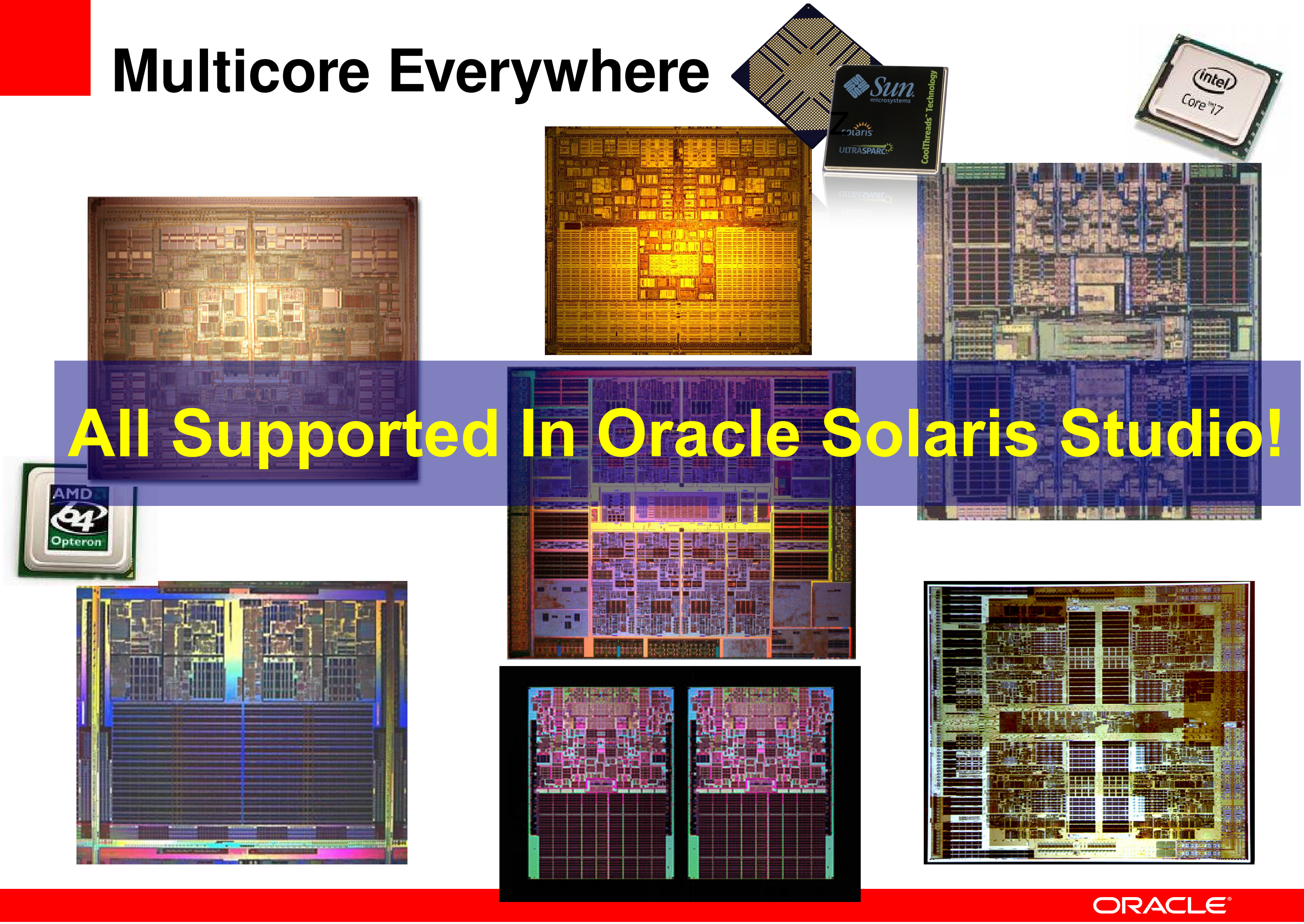


Memory Hierarchy Summary

- Caches play a crucial role in performance
- Today's memory hierarchy is very complex
  - Multicore processors have private and shared caches, often several levels
- Most important for technical computing:
  - Data cache(s)
  - TLB cache(s)
- Performance often depends on the problem size
- Accessing data in the wrong way can have a profound impact on performance

Oracle Solaris Studio

Multicore Everywhere



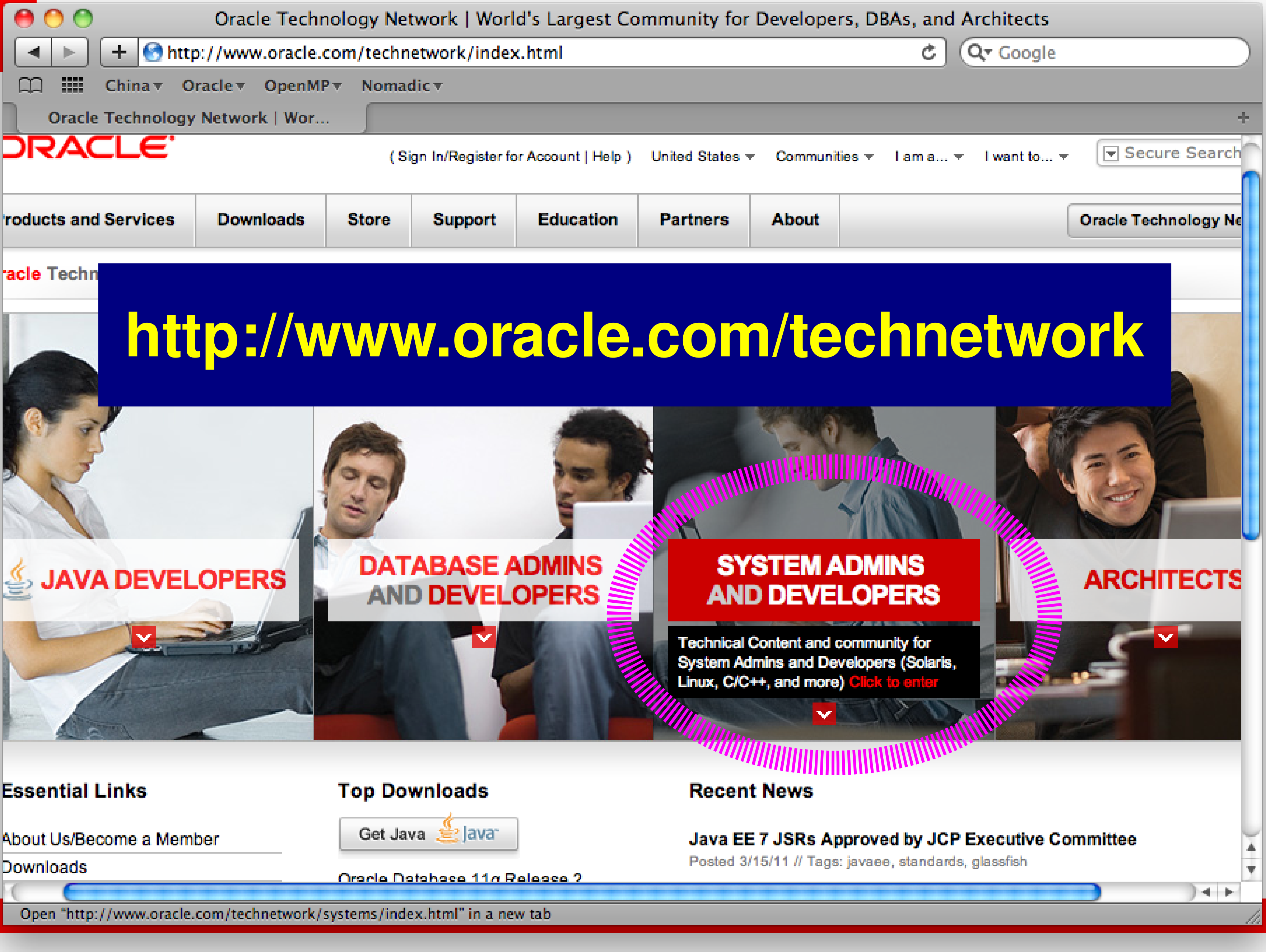
All Supported In Oracle Solaris Studio!

Oracle Solaris Studio/1

- Supports: Solaris on SPARC, Solaris/Linux on x86/x64
  - Processors supported: SPARC, Intel and AMD
- Fortran (f95), C (cc) and C++ (CC) compilers
  - Support state of the art sequential optimization
- Oracle Math Libraries
- Oracle Performance Library
- Automatic Parallelization
- OpenMP
- MPI (through the Oracle Message Passing Toolkit)
  - Based on Open MPI

Oracle Solaris Studio/2

- Performance Analyzer
  - Languages supported: Fortran, C, C++ and Java
  - Parallel: AutoPar, OpenMP, POSIX/Solaris Threads
    - MPI support greatly improved in Studio 12 Update 1
- Thread Analyzer
  - Languages supported: Fortran, C, C++
  - Parallel: OpenMP, POSIX/Solaris Threads
- Standalone GUI for dbx: dbxtool
- Studio IDE and other tools



http://www.oracle.com/technetwork

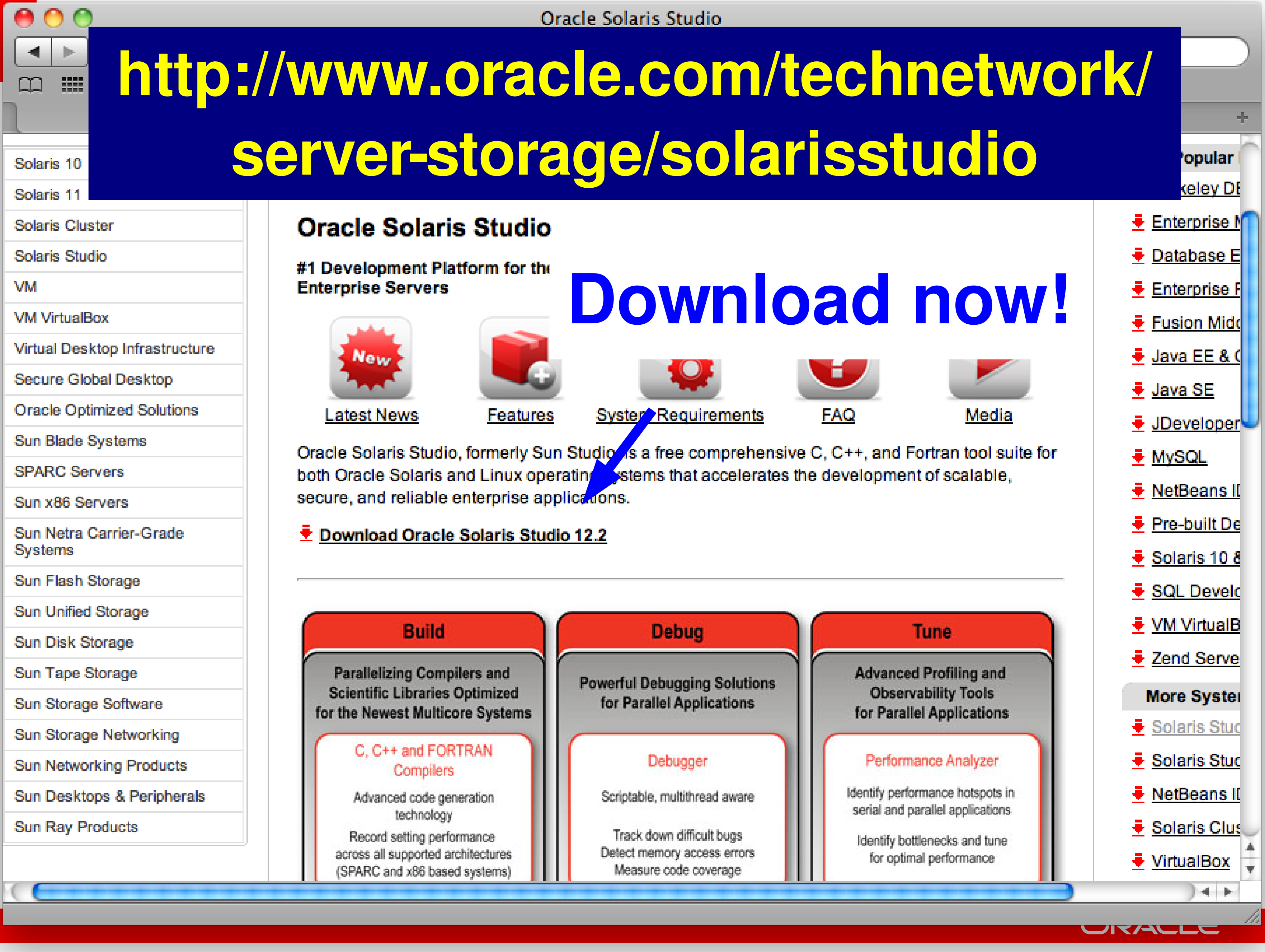
Oracle Technology Network | World's Largest Community for Developers, DBAs, and Architects
http://www.oracle.com/technetwork/index.html
Google
China
Oracle
OpenMP
Nomadic
Oracle Technology Network | Wor...
ORACLE
(Sign In/Register for Account | Help)
United States
Communities
I am a...
I want to...
Secure Search
roducts and Services
Downloads
Store
Support
Education
Partners
About
Oracle Technology Ne
racle Techn
JAVA DEVELOPERS
DATABASE ADMINS AND DEVELOPERS
SYSTEM ADMINS AND DEVELOPERS
ARCHITECTS

Technical Content and community for System Admins and Developers (Solaris, Linux, C/C++, and more) Click to enter

Java EE 7 JSRs Approved by JCP Executive Committee

Posted 3/15/11 // Tags: javaee, standards, glassfish

Open "http://www.oracle.com/technetwork/systems/index.html" in a new tab http://www.oracle.com/technetwork/server-storage/solarisstudio



Oracle Solaris Studio

- Solaris 10
- Solaris 11
- Solaris Cluster
- Solaris Studio
- VM
- VM VirtualBox
- Virtual Desktop Infrastructure
- Secure Global Desktop
- Oracle Optimized Solutions
- Sun Blade Systems
- SPARC Servers
- Sun x86 Servers
- Sun Netra Carrier-Grade Systems
- Sun Flash Storage
- Sun Unified Storage
- Sun Disk Storage
- Sun Tape Storage
- Sun Storage Software
- Sun Storage Networking
- Sun Networking Products
- Sun Desktops & Peripherals
- Sun Ray Products

Oracle Solaris Studio

#1 Development Platform for the Enterprise Servers

Download now!

Oracle Solaris Studio, formerly Sun Studio, is a free comprehensive C, C++, and Fortran tool suite for both Oracle Solaris and Linux operating systems that accelerates the development of scalable, secure, and reliable enterprise applications.

Download Oracle Solaris Studio 12.2

Build

Parallelizing Compilers and Scientific Libraries Optimized for the Newest Multicore Systems

C, C++ and FORTRAN Compilers

Advanced code generation technology

Record setting performance across all supported architectures (SPARC and x86 based systems)

Debug

Powerful Debugging Solutions for Parallel Applications

Debugger

Scriptable, multithread aware

Track down difficult bugs
Detect memory access errors
Measure code coverage

Tune

Advanced Profiling and Observability Tools for Parallel Applications

Performance Analyzer

Identify performance hotspots in serial and parallel applications

Identify bottlenecks and tune for optimal performance

Popular

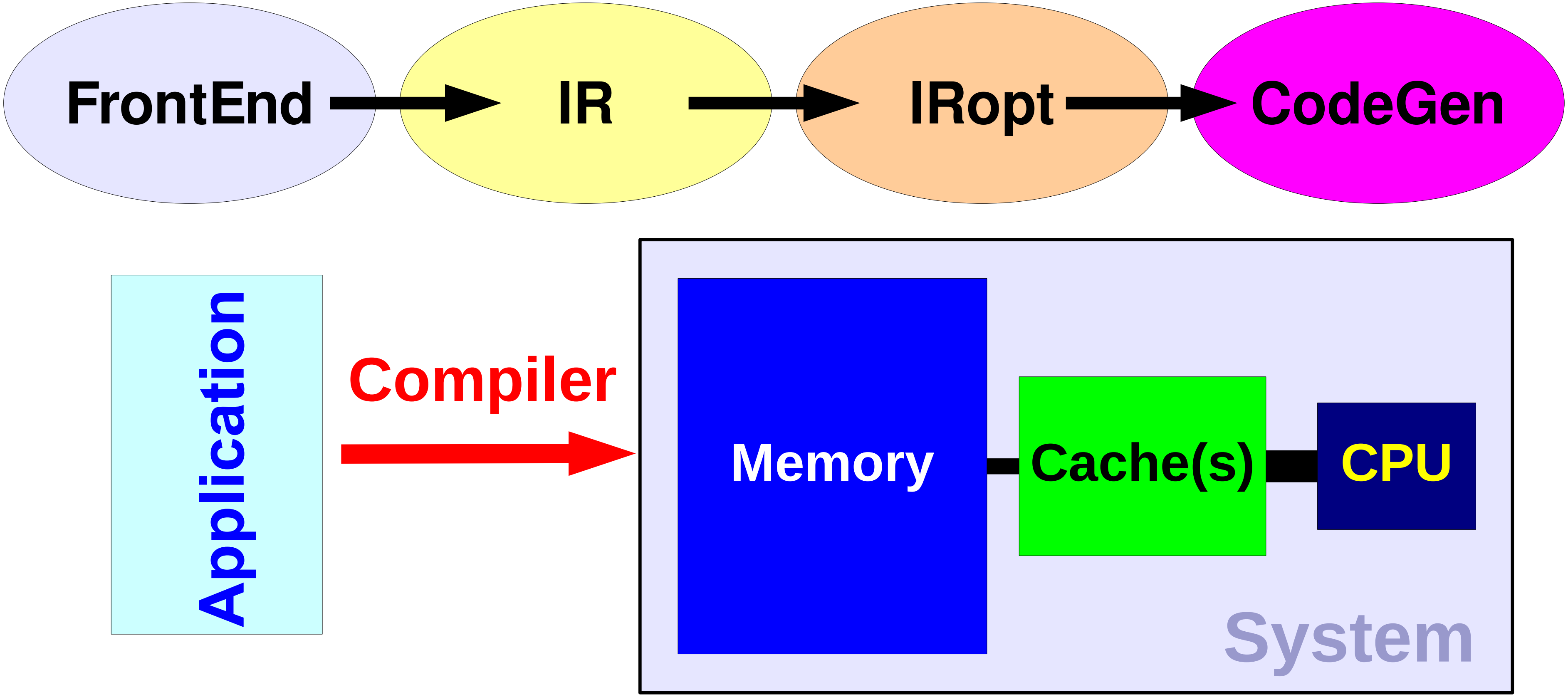
- keley DB
- Enterprise M
- Database E
- Enterprise P
- Fusion Midd
- Java EE & G
- Java SE
- JDeveloper
- MySQL
- NetBeans ID
- Pre-built De
- Solaris 10 &
- SQL Develo
- VM VirtualB
- Zend Serve

More System

- Solaris Stud
- Solaris Stud
- NetBeans ID
- Solaris Clus
- VirtualBox

Loop Based Optimizations

Who Does What ?



Introduction

- We now discuss several optimization techniques
  - Applicable to a variety of systems
  - Loop oriented
- All optimizations discussed are supported by the Oracle Solaris Studio compilers
  - Many more advanced optimizations are implemented
- In certain situations, different techniques can be used to achieve the same effect
- Code specific details and the underlying processor and/or system architecture may determine which one is best

The Compiler Commentary/1

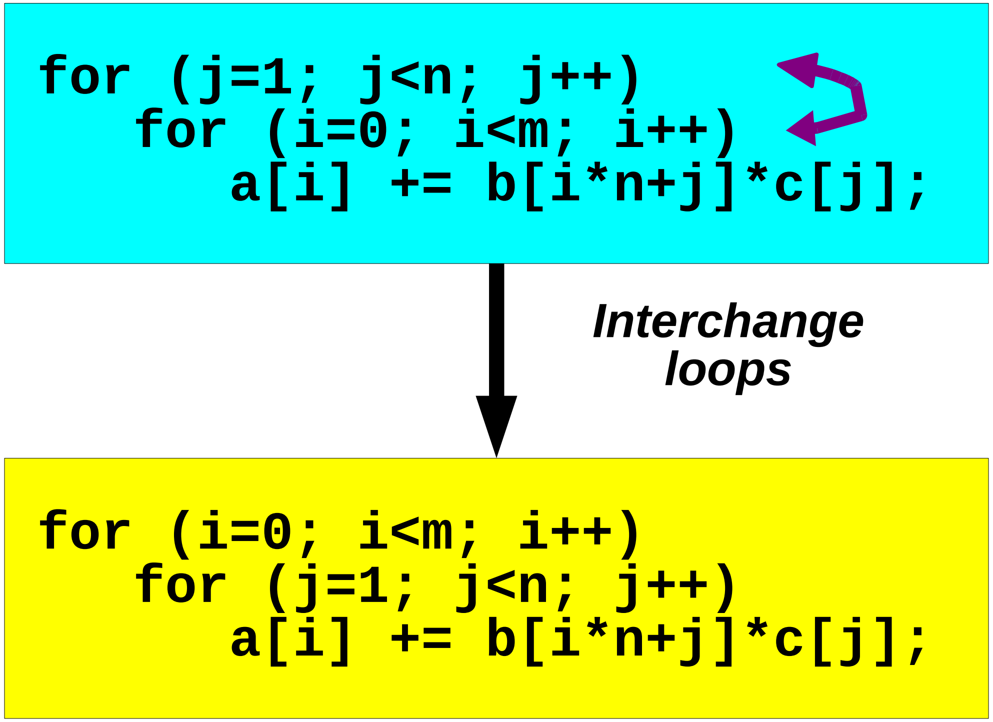
- The Oracle Solaris Studio compilers can generate information on the optimization and parallelization performed
  - One has to add the -g option (C and Fortran) or -g0 (C++) to the other compiler options to get these messages
- The Performance Analyzer displays the messages by default
- There is also the "*er_src*" command line tool to extract the information from the object file:

```
$ er_src my_object.o
```

The Compiler Commentary/2

- In the remaining part of this presentation we discuss generally applicable, but important, optimizations
- The example output shown has been obtained by using the "*er_src*" command

Loop Interchange



```
for (j=1; j<n; j++)
   for (i=0; i<m; i++)
      a[i] += b[i*n+j]*c[j];
```

Interchange loops

```
for (i=0; i<m; i++)
   for (j=1; j<n; j++)
      a[i] += b[i*n+j]*c[j];
```

- ***Vector “b” is accessed with non-unit stride***
- ***Interchanging the loops solves the problem***

Compiler Output

```
Source loop below has tag L1
L1 interchanged with L2
L1 scheduled with steady-state cycle count = 2
L1 unrolled 4 times
L1 has 2 loads, 1 stores, 3 prefetches, 1 FPadds,
0 FPmuls, and 0 FPdivs per iteration
L1 has 0 int-loads, 0 int-stores, 5 alu-ops, 0 muls,
0 int-divs and 0 shifts per iteration
 5.       for (j=0; j<n; j++)

Source loop below has tag L2
L2 interchanged with L1
 6.          for (i=0; i<m; i++)
 7.             a[i][j] = b[i][j] + c[i][j];
 8.
 9. }
```

Options: -fast -xrestrict

Loop Fission - Example

```
for (j=0; j<n; j++)
{
   c[j] = exp(j/n);
   for (i=0; i<m; i++)
      a[i][j]=b[i][j]+d[i]*e[j];
}
```

- ***Access on arrays 'a' and 'b' is bad***
- ***We can not simply interchange the loops***
- ***Fission/splitting is the solution***

Fission

```
for (j=0; j<n; j++)
   c[j] = exp(j/n);
```

New loop created

This loop can be further optimized

```
for (j=0; j<n; j++)
   for (i=0; i<m; i++)
      a[i][j]=b[i][j]+d[i]*e[j];
```

Interchange loops for better performance

Compiler Output

```
Source loop below has tag L1
L1 fissioned into 2 loops, generating: L3, L4
L3 interchanged with L2
L4 strip-mined by 1024, new inner loop L8
L3 cloned for unrolling-epilog.  Clone is L12
L8 fissioned into 1 loops, generating: L9
L8 transformed to use calls to vector intrinsics:
__vexp_
All 8 copies of L12 are fused together as part of unroll and jam
L12 scheduled with steady-state cycle count = 13
L12 unrolled 4 times
L12 has 9 loads, 8 stores, 16 prefetches, 8 FPadds, 8 FPmuls, and 0 FPdivs per iteration
L12 has 0 int-loads, 0 int-stores, 19 alu-ops, 0 muls, 0 int-divs and 0 shifts per iteration
L3 scheduled with steady-state cycle count = 2
L3 unrolled 4 times
L3 has 2 loads, 1 stores, 2 prefetches, 1 FPadds, 1 FPmuls, and 0 FPdivs per iteration
L3 has 0 int-loads, 0 int-stores, 5 alu-ops, 0 muls, 0 int-divs and 0 shifts per iteration
L9 scheduled with steady-state cycle count = 14
L9 unrolled 1 times
L9 has 1 loads, 1 stores, 0 prefetches, 1 FPadds, 0 FPmuls, and 0 FPdivs per iteration
L9 has 0 int-loads, 1 int-stores, 16 alu-ops, 0 muls, 0 int-divs and 3 shifts per iteration
      8.       for (j=0; j<n; j++)
      9.       {
     10.          c[j] = exp(j/n);

Source loop below has tag L2
L2 interchanged with L3
L2 cloned for unrolling-epilog.  Clone is L10
L10 is outer-unrolled 8 times as part of unroll and jam
     11.          for (i=0; i<n; i++)
     12.             a[i][j] = b[i][j] + d[i]*e[j];
     13.       }
```

Options: -fast -xrestrict -xvector

Loop Fusion - Example

```
for (i=0; i<n; i++)
    a[i] = 2 * b[i];
```

```
for (i=0; i<n; i++)
    c[i] = a[i] + d[i];
```

- ***Assume that 'n' is large***
- ***In the second loop, a[i] may no longer be in the cache***
- ***Fusing the loops ensures a[i] is still in the cache when needed***

Fusion

```
for (i=0; i<n; i++)
{
    a[i] = 2 * b[i];
    c[i] = a[i] + d[i];
}
```

Note that it is possible to apply fusion to loops with (slightly) different boundaries

In such a case, some iterations need to be 'peeled' off

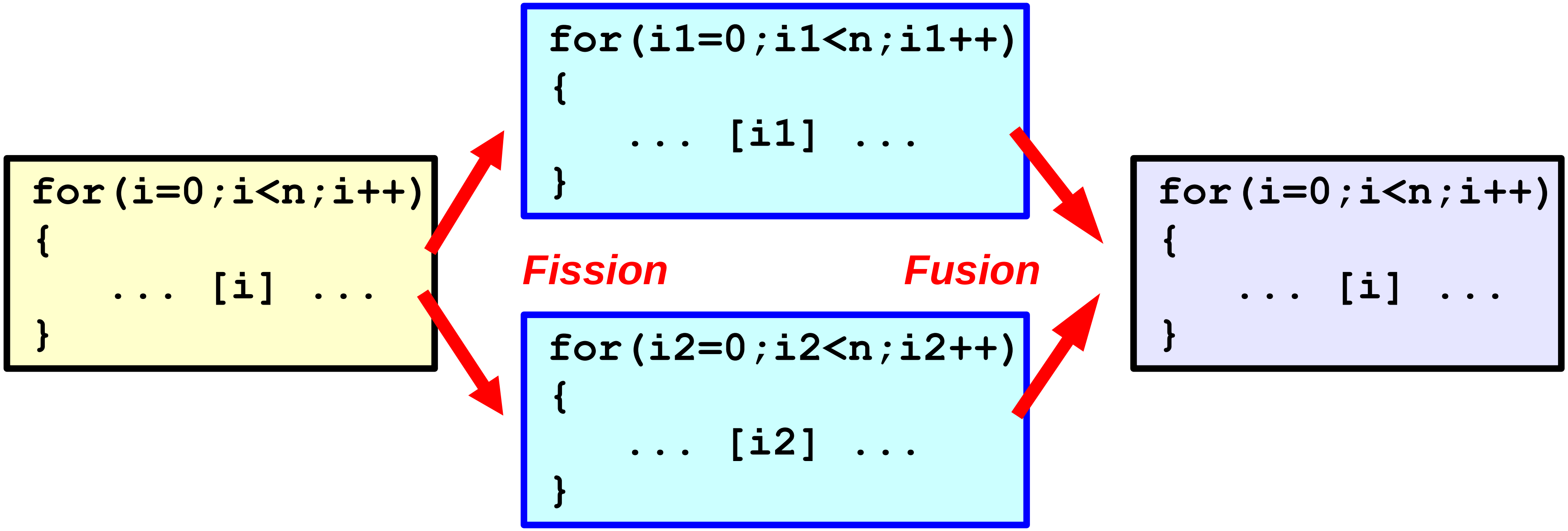
Compiler Output

```
Source loop below has tag L1
L1 fused with L2, new loop L3
L3 scheduled with steady-state cycle count = 3
L3 unrolled 4 times
L3 has 2 loads, 2 stores, 4 prefetches, 2 FPadds, 0
FPmuls, and 0 FPdivs per iteration
L3 has 0 int-loads, 0 int-stores, 6 alu-ops, 0 muls, 0
int-divs and 0 shifts per iteration
     5.      for (i=0; i<n; i++)
     6.         a[i] = 2 * b[i];

Source loop below has tag L2
     7.      for (i=0; i<n; i++)
     8.         c[i] = a[i] + d[i];
```

Options: -fast -xrestrict

Loop Fission and Fusion



```
for(i=0;i<n;i++)
{
    ... [i] ...
}
```

```
for(i1=0;i1<n;i1++)
{
    ... [i1] ...
}
```

```
for(i2=0;i2<n;i2++)
{
    ... [i2] ...
}
```

```
for(i=0;i<n;i++)
{
    ... [i] ...
}
```

Fission

- ✓ *Enable loop interchange*
- ✓ *Isolate dependences*
- ✓ *Increase opportunities for optimization (e.g. vectorization of intrinsics)*
- ✓ *Reduce register pressure*

Fusion

- ✓ *Reduce cache reloads*
- ✓ *Increase Instruction Level Parallelism (ILP)*
- ✓ *Reduce loop overhead*

Inner Loop Unrolling - Example

Through unrolling, the loop overhead ('book keeping') is reduced

```
for (i=0; i<n; i++)
   a[i] = b[i] + c[i];
```

```
Loads       : 2
Stores      : 1
FP Adds     : 1
I=I+1
Test I < N ?
Branch
Addr. incr: 3
```

Work: 4
Overhead: 6

Loop is unrolled with a factor of 4

```
for (i=0; i<n-n%4; i+=4)
{
   a[i  ] = b[i  ] + c[i  ];
   a[i+1] = b[i+1] + c[i+1];
   a[i+2] = b[i+2] + c[i+2];
   a[i+3] = b[i+3] + c[i+3];
}
// This is the clean up loop
for (i=n-n%4; i<n; i++)
   a[i] = b[i] + c[i];
```

```
Loads       : 8
Stores      : 4
FP Adds     : 4
I=I+4
Test I < N ?
Branch
Addr. incr: 3
```

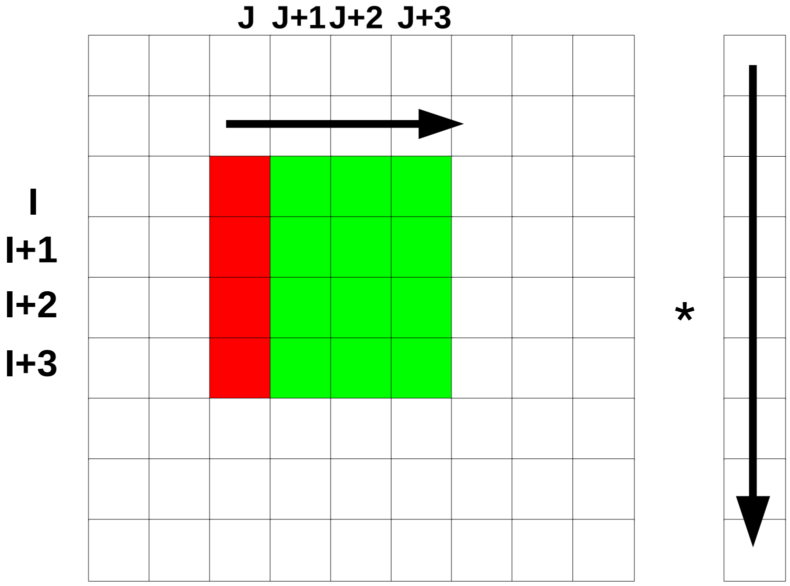
Work: 16
Overhead: 6

Compiler Output

```
Source loop below has tag L1
L1 scheduled with steady-state cycle count = 2
L1 unrolled 4 times
L1 has 2 loads, 1 stores, 3 prefetches, 1 FPadds, 0 FPmuls,
and 0 FPdivs per iteration
L1 has 0 int-loads, 0 int-stores, 5 alu-ops, 0 muls, 0 int-
divs and 0 shifts per iteration
     5.      for (i=0; i<n; i++)
     6.         a[i] =  b[i] + c[i];
```

Options: -fast -xrestrict

Outer Loop Unrolling - Example

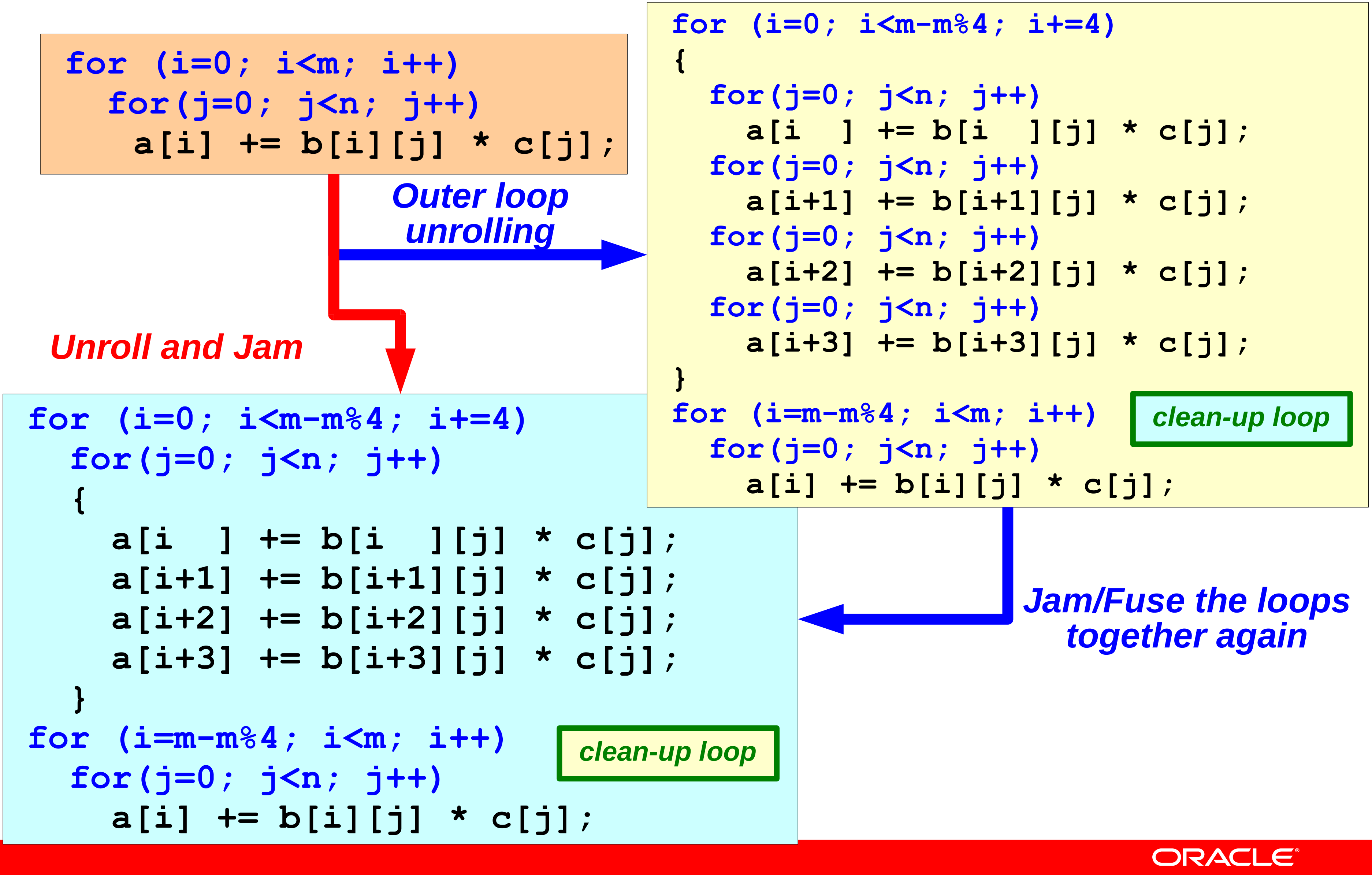


```
for (i=0; i<m; i++)
   for(j=0; j<n; j++)
   {
      a[i] += b[i][j] * c[j];
   }
```

```
for (i=0; i<m; i+=4)
   for(j=0; j<n; j++)
   {
      a[i  ] += b[i  ][j] * c[j];
      a[i+1] += b[i+1][j] * c[j];
      a[i+2] += b[i+2][j] * c[j];
      a[i+3] += b[i+3][j] * c[j];
   }
<clean-up loop>
```

- ***Advantage: re-use of c[j] (temporal locality)***
- ***Deeper unrolling requires more registers, but improves re-use of c[j]***

Unroll and Jam



```
for (i=0; i<m; i++)
   for(j=0; j<n; j++)
     a[i] += b[i][j] * c[j];
```

Outer loop unrolling

```
for (i=0; i<m-m%4; i+=4)
{
   for(j=0; j<n; j++)
     a[i  ] += b[i  ][j] * c[j];
   for(j=0; j<n; j++)
     a[i+1] += b[i+1][j] * c[j];
   for(j=0; j<n; j++)
     a[i+2] += b[i+2][j] * c[j];
   for(j=0; j<n; j++)
     a[i+3] += b[i+3][j] * c[j];
}
for (i=m-m%4; i<m; i++)
   for(j=0; j<n; j++)
     a[i] += b[i][j] * c[j];
```

clean-up loop

Jam/Fuse the loops together again

Unroll and Jam

```
for (i=0; i<m-m%4; i+=4)
   for(j=0; j<n; j++)
   {
     a[i  ] += b[i  ][j] * c[j];
     a[i+1] += b[i+1][j] * c[j];
     a[i+2] += b[i+2][j] * c[j];
     a[i+3] += b[i+3][j] * c[j];
   }
for (i=m-m%4; i<m; i++)
   for(j=0; j<n; j++)
     a[i] += b[i][j] * c[j];
```

clean-up loop

Compiler Output

```
Source loop below has tag L1
L1 cloned for unrolling-epilog.  Clone is L3
L3 is outer-unrolled 8 times as part of unroll and jam
     5.     for (i=0; i<m; i++)

Source loop below has tag L2
L2 cloned for unrolling-epilog.  Clone is L5
All 8 copies of L5 are fused together as part of unroll and jam
L2 scheduled with steady-state cycle count = 2
L2 unrolled 4 times
L2 has 2 loads, 0 stores, 1 prefetches, 1 FPadds, 1 FPmuls, and
0 FPdivs per iteration
L2 has 0 int-loads, 0 int-stores, 4 alu-ops, 0 muls, 0 int-divs
and 0 shifts per iteration
L5 scheduled with steady-state cycle count [illegible]
L5 unrolled 4 times
L5 has 9 loads, 0 stores, 8 prefetches, 8 FPadds, 8 FPmuls, and
0 FPdivs per iteration
L5 has 0 int-loads, 0 int-stores, 11 alu-ops, 0 muls, 0 int-divs and
0 shifts per iteration
      6.          for (j=0; j<n; j++)
      7.               a[i] += b[i][j]*c[j];
```

outer loop unrolling

inner loop unrolling

inner loop unrolling

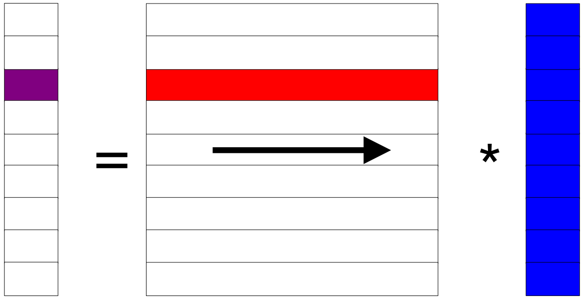
Options: -fast -xrestrict

Matrix Times Vector in Fortran

Two implementations of matrix times vector:

Row-wise access

```
DO I = 1, N
   DO J = 1, N
      A(I) = A(I) + B(I,J)*C(J)
   END DO
END DO
```



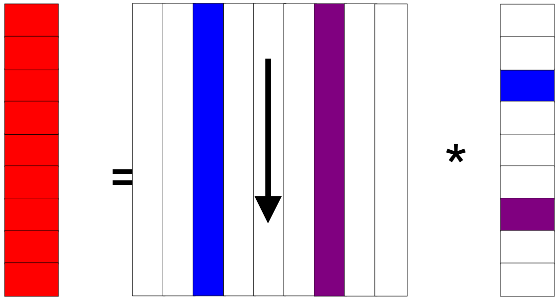
Bad Memory Access −

2 Loads +

Loop Interchange

Column-wise access

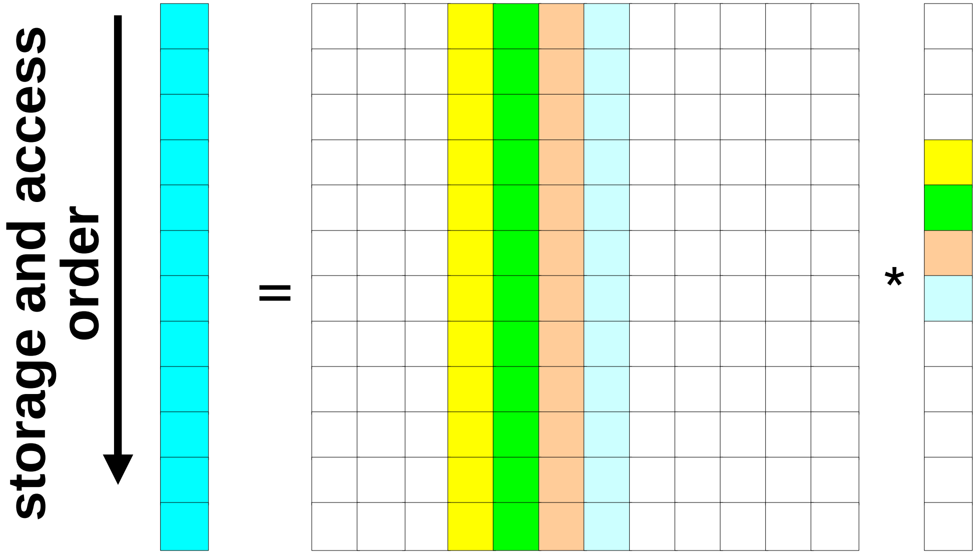
```
DO J = 1, N
   DO I = 1, N
      A(I) = A(I) + B(I,J)*C(J)
   END DO
END DO
```



Good Memory Access +

2 Loads and 1 Store −

Column Version In Fortran



Unrolling over J:

- Good cache line utilization
- Good TLB performance
- Reduce loads/stores on A

```
DO J = 1, N, 4
   DO I = 1, N
      A(I) = A(I) + B(I,J  )*C(J  )  + B(I,J+1)*C(J+1)+
                    B(I,J+2)*C(J+2)  + B(I,J+3)*C(J+3)
   END DO
END DO
<clean up loop for J>
```

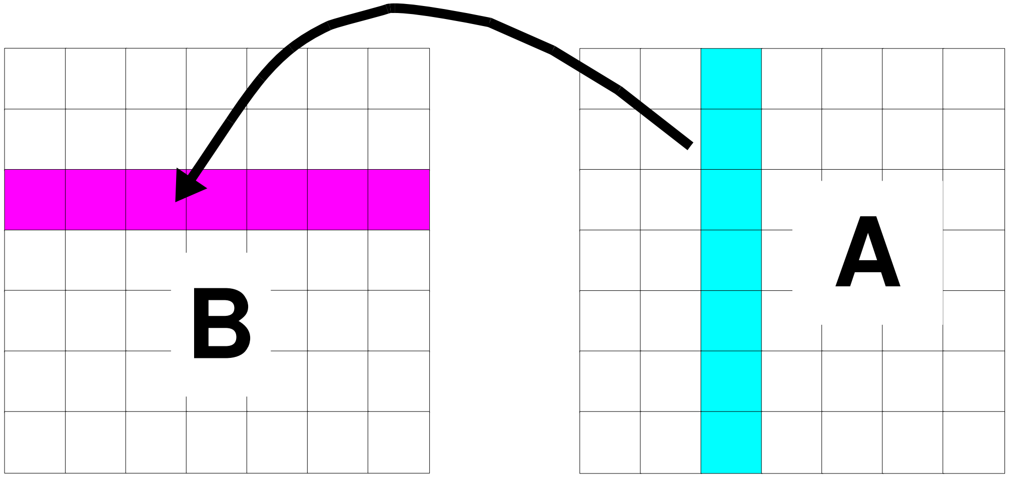
Loop Unrolling - Summary

- Execute more than one iteration per loop pass
- Inner loop unrolling advantages:
  - Reduce loop overhead
  - Better instruction scheduling
- Outer loop unrolling advantages:
  - Improve cache line usage (spatial locality)
  - Re-use data (temporal locality)
- Disadvantages of unrolling:
  - More registers needed
  - Clean-up code required

Loop Blocking - Example/1

Transposing a matrix

```
for (j=0; j<n; j++)
   for (i=0; i<n; i++)
      b[j][i] = a[i][j];
```

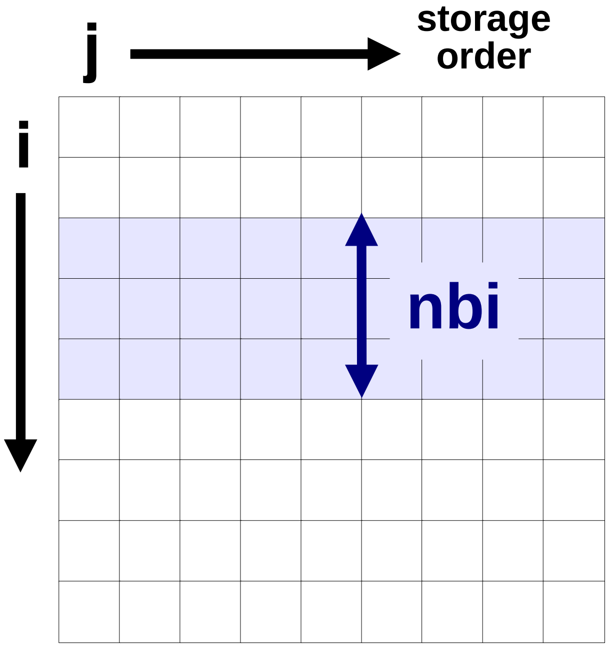


- *Loop interchange does not help here:*
  - *Role of 'a' and 'b' is only going to be interchanged*
- *Change of programming language won't help either*
- *Unrolling the i-loop can be beneficial, but requires more registers and doesn't address TLB-misses*
- *Loop blocking achieves good memory performance, without the need for additional registers*

Loop Blocking - Example/2

Blocking and interchanging the I-loop

```
for(i1=0; i1<n; i1+=nbi)
   for (j=0; j<n; j++)
       for (i2=0;i2<MIN(n-i1,nbi);i2++)
           b[j][i1+i2] = a[i1+i2][j];
```



- ***Parameter 'nbi' is the blocking size***
- ***Should be chosen as large as possible***
  - ✔ ***A too short loop may cause stall***
- ***Actual value depends on the size of the target cache to block for***
  - ✔ ***The D-TLB could be a target too***

Fortran

```
do i = 1, n
```

```
do i1 = 1, n, nbi
  do i2 = 0,min(n-i1+1,nbi)-1
```

Loop Blocking - Summary

- Powerful technique to improve:
  - Memory access (spatial locality)
  - Data re-use (temporal locality)
- Preserves portability, but blocking size depends on:
  - Cache type/level/capacity
  - Data requirements
- Recommendations:
  - Choose blocking size as large as possible
  - Leave some space for other data
  - Parameterize cache characteristics, especially size

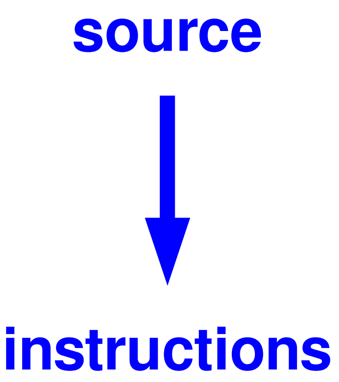
Optimization Benefits

Optimization	Oracle Solaris Studio Compiler	Instruction	Memory
Loop Interchange	yes	+	++
Loop Fission	yes	+	++
Loop Fusion	yes	+	++
Inner Loop Unrolling	yes	++	-
Outer Loop Unrolling	yes	+	++
Loop Blocking	yes	-	++

Instruction Scheduling Optimizations

From Source To Instructions

```
double average(int n, double data[])
{
  double sum = 0.0;
  for (int i=0; i<n; i++)
      sum += data[i];
  return(sum/n);
}
```


source

instructions

```
                      ......
805136c:  addl      $-1,%ecx
805136f:  movl      0x24(%esp),%edx
8051373:  movl      %eax,%esi
8051375:  jns       .+4 [ 0x8051379 ]
8051377:  xorl      %esi,%esi
8051379:  cmpl      $8,%esi
805137c:  jl        .+0x41 [ 0x80513bd ]
805137e:  addl      $-8,%eax
8051381:  prefetcht0  0x100(%edx)
8051388:  addsd     (%edx),%xmm1
            ......
```

Superscalar Execution

- ***N-way superscalar:***
  - ***Execute N instructions at the same time***
- ***This is also called Instruction Level Parallelism (ILP)***

	slot 1	slot 2	slot 3	slot 4	
cycle 1					4-way superscalar
cycle 2	not used				3-way superscalar
cycle 3		not used	not used		2-way superscalar
cycle 4	not used			not used	2-way superscalar
cycle 5			not used		3-way superscalar

- ***This is a transparent hardware feature, but by carefully scheduling the instructions, optimizing compilers can greatly enhance performance***

An Example - Single Issue Execution

```
for (i=0; i<n; i++)
   x[i] = x[i] + c*y[i];
```

Instruction	Cycle				
	1	2	3	4	5	6
load	r1=y[0]					r1=y[1]
load		r2=x[0]				
multiply			r3=c*r1			
add				r4=r2+r3		
store					r4	

One result takes 5 cycles to compute

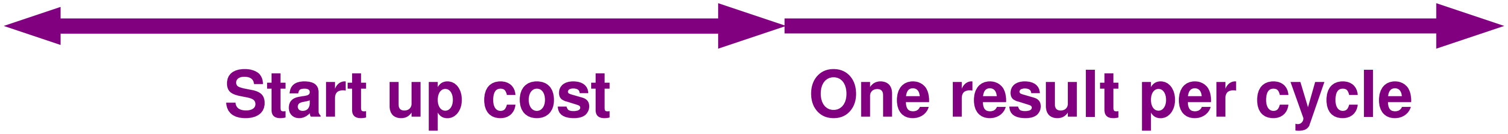
Imaginary Processor Characteristics

- Simplified Assumptions:
  - Each instruction takes one cycle to complete
  - A load instruction takes a single cycle only
- Superscalar features:
  - 5-way superscalar
  - The following instructions can be issued simultaneously
    - Two loads, one store
    - A floating-point multiply and add

An Example - Superscalar Execution

```
for (i=0; i<n; i++)
   x[i] = x[i] + c*y[i];
```

Instruction	Cycle					
	1	2	3	4	5	6
load	r1=y[0]	r3=y[1]	r1=y[2]	r3=y[3]	r1=y[4]	......
load	r2=x[0]	r4=x[1]	r6=x[2]	r2=x[3]	r4=x[4]	......
multiply		r5=c*r1	r7=c*r3	r5=c*r1	r7=c*r3	......
add			r8=r2+r5	r9=r4+r7	r8=r6+r5	......
store				r8	r9	r8



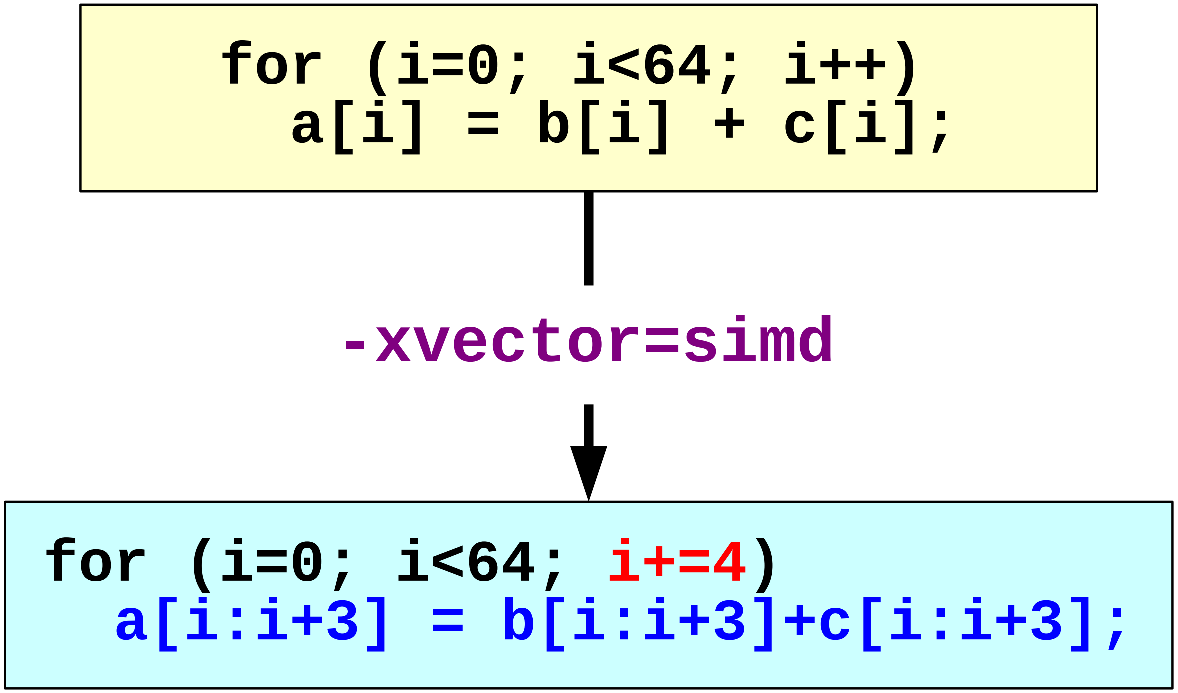
Start up cost

One result per cycle

SIMD Instructions

- Special “vector” instructions simultaneously process multiple data elements in adjacent memory
  - Often called SIMD (“*Single Instruction, Multiple Data*”)
- Best suited for processing “streaming data”
  - For example array operations in a loop
- The Oracle Solaris Studio compilers can generate such SIMD instructions
  - Use the `-xvector=simd` option for this
- Can use the `-xarch` option to specify the instruction set
  - For example `-xarch=sse3` (Intel/AMD) or `-xarch=sparcvis3` (SPARC)

Example Vector Instructions (SIMD)



```
for (i=0; i<64; i++)
  a[i] = b[i] + c[i];
```

-xvector=simd

```
for (i=0; i<64; i+=4)
  a[i:i+3] = b[i:i+3]+c[i:i+3];
```

You write this

The compiler generates the appropriate code

One single instruction updates 4 elements in parallel !

Instruction Scheduling Optimization

- Instruction level optimizations mostly come into play:
  - If memory access cost is not an issue (any longer)
- The Oracle Solaris Studio compilers aggressively optimize the instruction schedule for the target processor
  - Exploit the architecture specific superscalar features
  - Deal with various instruction latencies
  - Various other low level optimizations
- What can <u>you</u> do?
  - Select the right instruction set - Typically the most recent set is also the most powerful
  - Use 64-bit addressing (-m64 option) on Intel/AMD in particular

Performance Considerations

The Big Picture

- Powerful compiler optimizations are available
- The compiler is however limited by:
  - Lack of knowledge of the application
  - Your data structures
- It is up to the developer to write code such that the compiler can find opportunities for optimization
- Structure the code appropriately
- Leave the low level details to the compiler

Best Practices/1

- Use a tool to tell you where the time is spent
  - Oracle Solaris Studio has the Performance Analyzer for this
- Split source in compute intensive part + “the rest”
  - To optimize the compute part more aggressively
- Look into using optimized libraries
- Use the Compiler Commentary to verify what the compiler did
- Write efficient, but clear code
- Avoid very "fat"/”bulky” loops

Best Practices/2

- Minimize the use of global data
- Branches (if-then-else):
  - Simplify where possible
  - May be beneficial to split the branch part out of loop
  - Consider profile feedback if supported on your compiler
    - Oracle Solaris Studio has the -xprofile option to do this

Best Practices/3 - Oracle Solaris Studio

- Compile and link with the `-fast` option
  - Single option meant to give good performance across a wide range of applications
  - Takes architecture specific settings (e.g. cache sizes) from compile platform
    - Use options like `-xtarget` and/or `-xarch` to override
- Experiment with some additional options
  - Prefetch: `-xprefetch` and `-xprefetch_level`
  - On Intel and AMD: consider 64-bit addressing (`-m64` option)
  - Exploit SIMD instructions (`-xvector=simd` option)
  - Use the `-xarch` option to select a non-default instruction set
    - Typically, a more recent instruction set is more powerful

A Simple But Effective Application Tuning Strategy

- Port the program
- Use the Oracle Performance Analyzer to find:
  - The part(s) where most of the time is spent
- Find the best set of compiler options
  - For example, experiment with prefetch options
- For the time consuming parts, make sure memory access is optimal
  - If this can not be fixed, try large pages
- Check the messages from the compiler (e.g “er_src”)
  - Experiment with options to get the desired behavior
  - If needed, consider to modify the source

Thank You And .... Stay Tuned !

ruud.vanderpas@oracle.com

Hardware and Software

Engineered to Work Together

ORACLE®