IBM Power Systems Compiler Strategy

Roch Archambault
IBM Toronto Laboratory
archie@ca.ibm.com

IBM Rational Disclaimer

Agenda

- IBM Compiler Team
- Collaboration with US Government
- Overall Roadmap
- Common Features and Compiler Architecture
- Key Technology Exploitation
  - Fortran 2003
  - OpenMP 3.0
  - UPC Compiler
  - CAF Compiler
  - OpenCL
  - Delinquent Load Analysis
  - Assist Threads For Prefetching
  - Compiler Transformation Report Infrastructure
  - Automatic Parallelization
- Q&A

IBM Compilation Technology Group

More than 300 development, test and service engineers. Mostly in Toronto, Canada

Product responsibility for high performance C, C++ and Fortran compilers and Math libraries (MASS,MASSV) targeting IBM servers/CPUs.

Responsible for Java JIT compilers targeting handheld devices to 64-way servers and everything in between.

Also develops compilers for commercial markets

C/C++ (pSeries and zSeries), COBOL, PLX for zSeries

All in-house technology developed over the past 25 years in close conjunction with IBM Research and HW teams in Austin and Rochester.

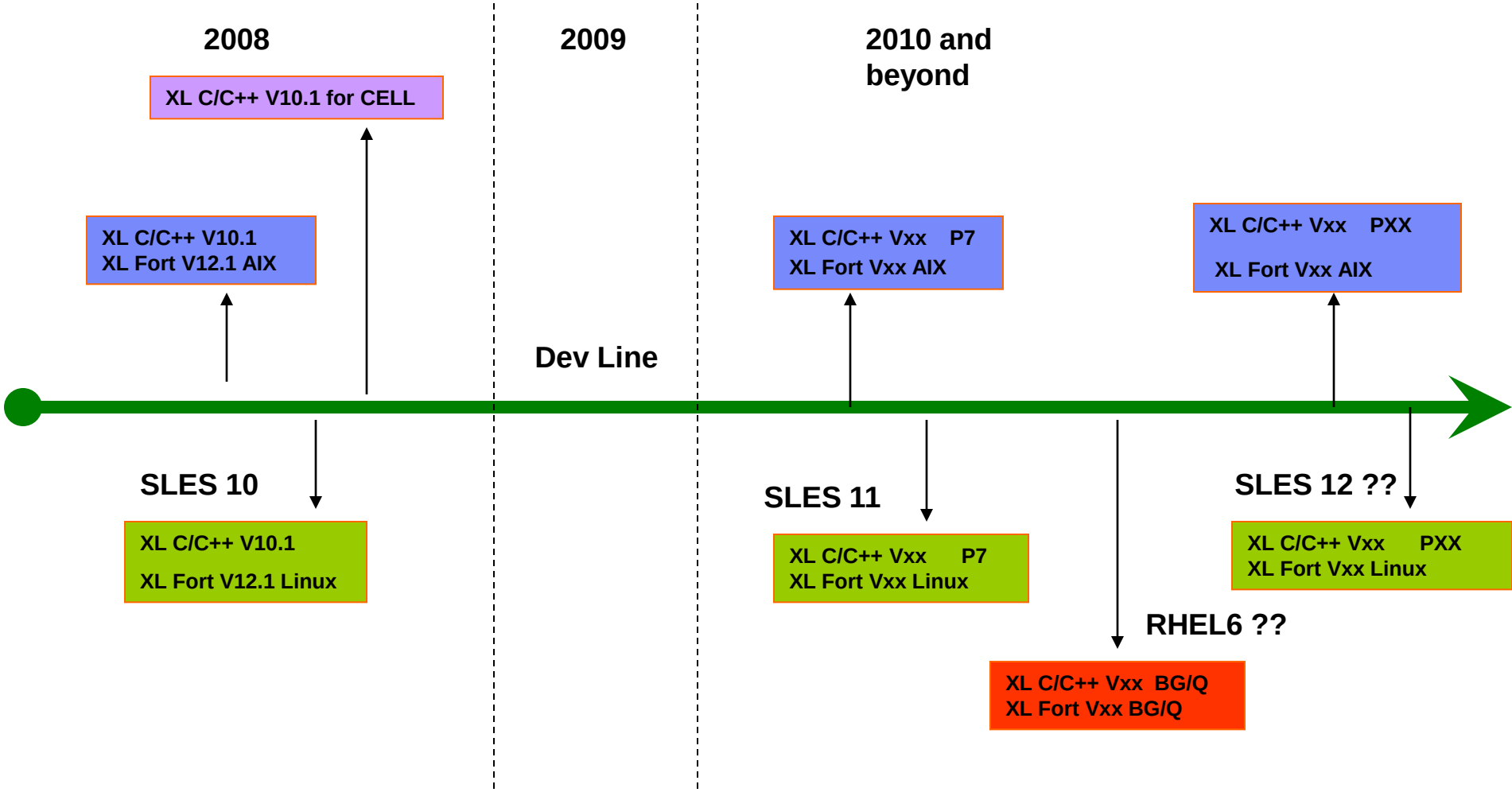
Our Mission:

To deliver the highest performance, most robust, most up-to-date language implementations in support of IBM's Hardware, Software and Services businesses

Collaboration with US Government

- Involved in first system to achieve 1 TF sustained (sPPM) in late 1990s (ASCI Blue)
  - Toronto compiler group developed automatic vectorization technology (calls to vector math routines) in collaboration with LLNL.
  - Continued support of ASCI program with White and Purple.
- ASC collaboration with Blue Gene and Roadrunner systems
- HPCS collaboration with delivery of UPC and programmer productivity improvements.
- Bluewater collaboration with delivery of CAF product and scaling challenges.
- Sequoia collaboration with delivery of compiler to exploit BG/Q features.

Roadmap of XL Compiler Releases

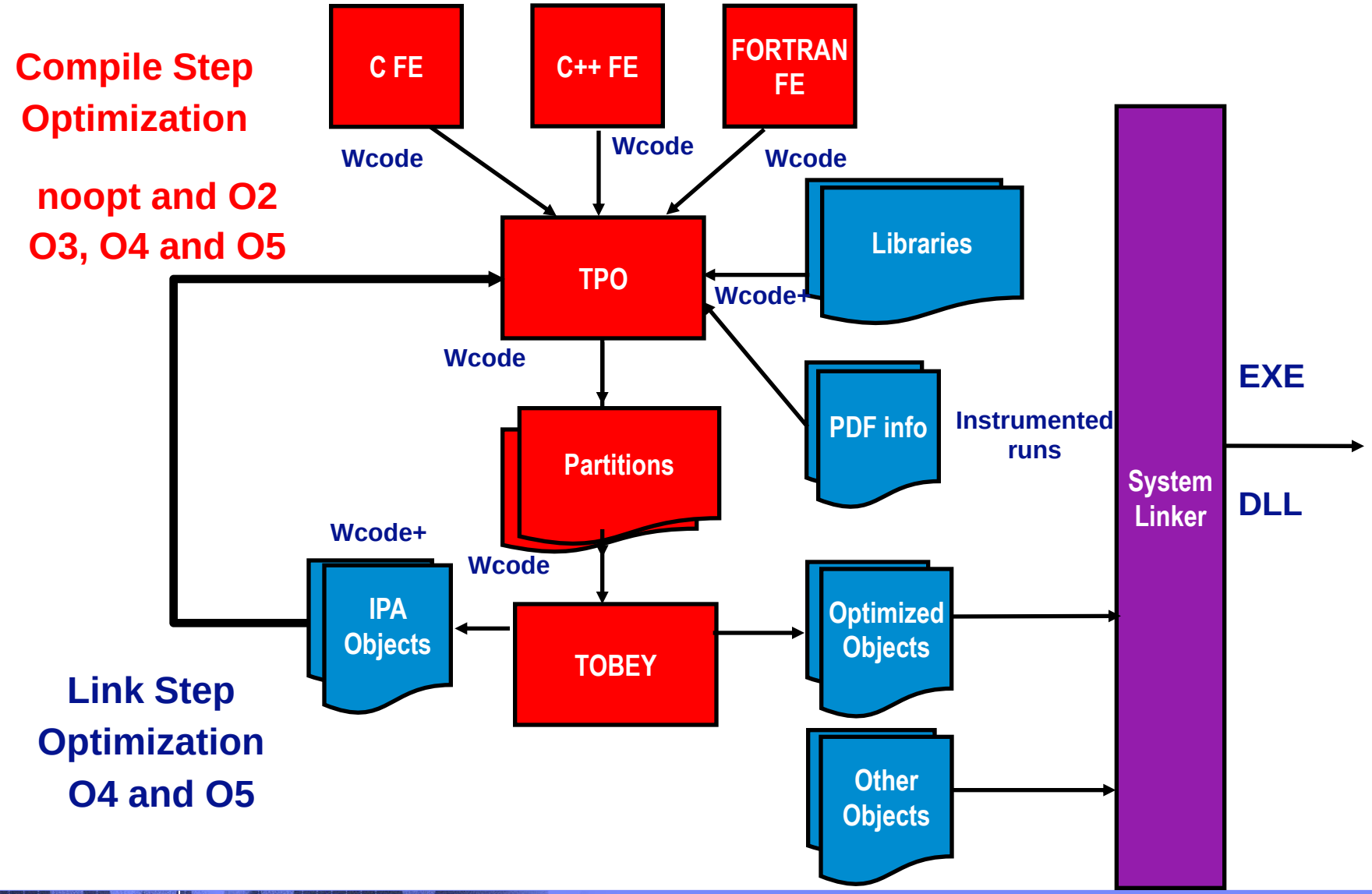


All information subject to change without notice

Common Fortran, C and C++ Features

- Linux (SLES and RHEL) and AIX, 32 and 64 bit
- Debug support
  - Debuggers on AIX:
    - Total View (TotalView Technologies), DDT (Allinea), IBM Debugger and DBX
  - Debuggers on Linux:
    - TotalView, DDT and GDB
- Full support for debugging of OpenMP programs (TotalView)
- Snapshot directive for debugging optimized code
- Portfolio of optimizing transformations
  - Instruction path length reduction
  - Whole program analysis
  - Loop optimization for parallelism, locality and instruction scheduling
  - Use profile directed feedback (PDF) in most optimizations
- Tuned performance on POWER3, POWER4, POWER5, PPC970, PPC440, PPC450, POWER6, POWER7 and CELL systems
- Optimized OpenMP

IBM XL Compiler Architecture



Compiler support for performance and productivity

Highest performance with programmer control

Iterative compilation

User reports, PDF

Manual Intervention

Analyse, change,

SIMD/Alignment ..

recompile

Parallel Languages

Semi-automatic

UPC, CAF,

OpenMP, OpenCL

Automatic Optimization

Compiler supplied

Memory optimization

Auto-simd

Auto-parallel

Highest Productivity with fully automatic compiler technology

Key Technology Exploitation

Emerging Standards and Tools Related Development:

- More Fortran 2003 support (Parameterized Derived Types)
- More OpenMP 3.0 support in Fortran
- Support for C++ 0x Standard
- Support for Fortran 2008 Standard
- Support for UPC language
- Support for Coarray Fortran language
- Support for OpenCL language
- Support for Propolice (Stack smashing protection in C)
- Support for POMP (OpenMP performance analysis)

Compiler Infrastructure and Optimization:

- Compiler optimization report in XML
- PDF (Profile Directed Feedback) extension:
  - Multi pass profiling
  - Delinquent load analysis and optimizations
- Assist threads for prefetching
- Polyhedral loop transformations
- Automatic Parallelization
- Analysis of MPI applications

All information subject to change without notice

Fortran 2003: Object Oriented Extensions

type extension (inheritance)

```
type fluid
  real :: viscosity
real, allocatable :: velocity(:,:,:)
end type

type, extends(fluid) :: magnetofluid
  real, allocatable :: magnetic_field(:,:,:)
end type
```

- type magnetofluid inherited ALL of properties of fluid: viscosity and velocity
- Only support single-rooted inheritance hierarchy

Fortran 2003: Object Oriented Extensions

type-bound procedures

```
  type point
    real x, y
    contains
      procedure :: length => lenBetween2Points
  end type
...!definition of lenBetween2Points
real function lenBetween2Points(this, p)
    class(point), intent(in) :: this, p
    ... ! compute the length
end function
...! in main program
  type(point) :: pa, pb
  ...
distance = pa%length(pb)
```

Fortran 2003: Object Oriented Extensions

Work in Progress

derived type allowed to have KIND and LENGTH type parameters

```
integer, parameter::dp = selected_real_kind(15)
  type matrix(kind,m,n)
    integer, kind :: kind=dp
    integer, len :: m, n
    real(kind) :: element(m,n)
  end type

  type(matrix(dp,10,20)) :: a
```

declares a double-precision matrix of size 10 x 20

```
type(matrix(dp, :, :)), allocatable :: mat
...
ALLOCATE (matrix(dp, m, n) :: mat)
```

size of matrix mat is determined at runtime

OpenMP 3.0: Task Support

irregular parallelism

a task has

- code to execute
- a data environment (it owns its data)
- an assigned thread executes the code and uses the data

two activities: packaging and execution

- each encountering thread packages a new instance of a task (code and data)
- some thread in the team executes the task

task construct

- defines an explicit task
- directive: task / end task
- clause: if, untied, private, firstprivate, default, and shared

task switching

- the act of a thread to switch from executing one task to another task

task scheduling point

- a point during the execution of the current task region at which it can be suspended to be resumed later; or the point of task completion, after which the executing thread may switch to a different task region
- e.g. encountered task constructs, encountered taskwait constructs

OpenMP 3.0: Task Support

generate independent works with task construct

```
!$OMP parallel
!$OMP single
      do while (...)
!$OMP task
        call process(p)
!$OMP end task
      enddo
!$OMP end single
!$OMP end parallel
```

XL UPC Compiler

Tech preview on alphaWorks

- Based on XL C V10.1 compiler
- Compiler generated interface to the runtime system is identical for shared and distributed memory implementations
- Optimizations take advantage of system architecture knowledge

On AIX

- Shared Memory (pthreads)
- Distributed (LAPI)

On Linux

- Shared Memory (pthreads)
- Distributed (LAPI)

On BG/L

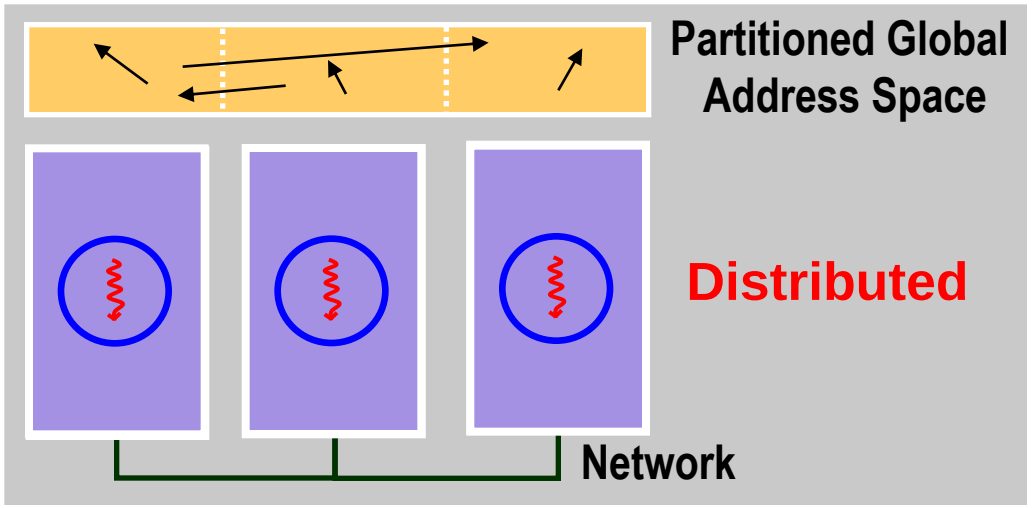
- BG Message Layer (based on XLC V9.1 compiler)

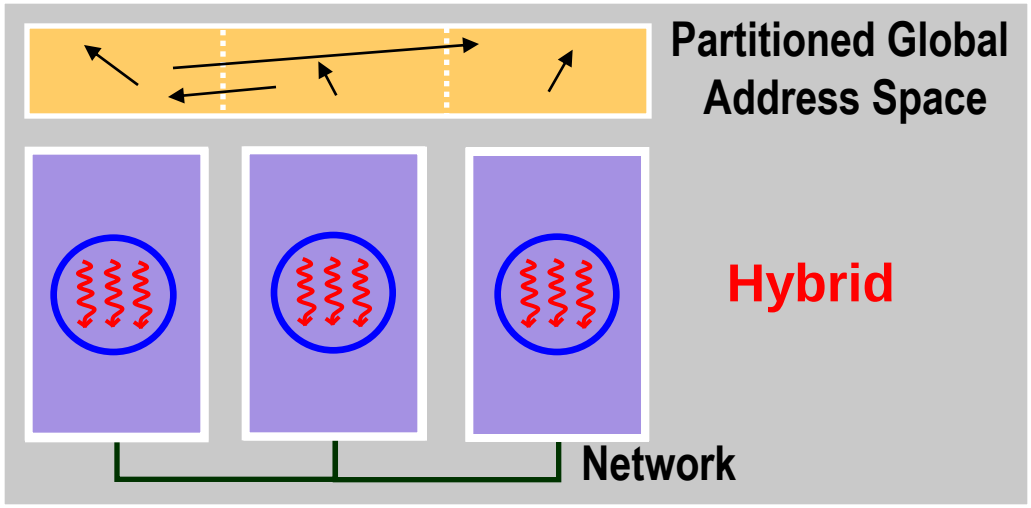
Using approximately 1000 test scenarios:

- GWU UPC test suite
- UPC version of NAS benchmarks
- Berkeley UPC test suite
- MTU UPC test suite
- HPC Challenge suite

Hybrid Execution Environment

Thread Process Node





Execution Environments

- Distributed: multiple nodes, one (or more) process per node running one thread
- Hybrid: multiple nodes, one (or more) processes per node running multiple threads

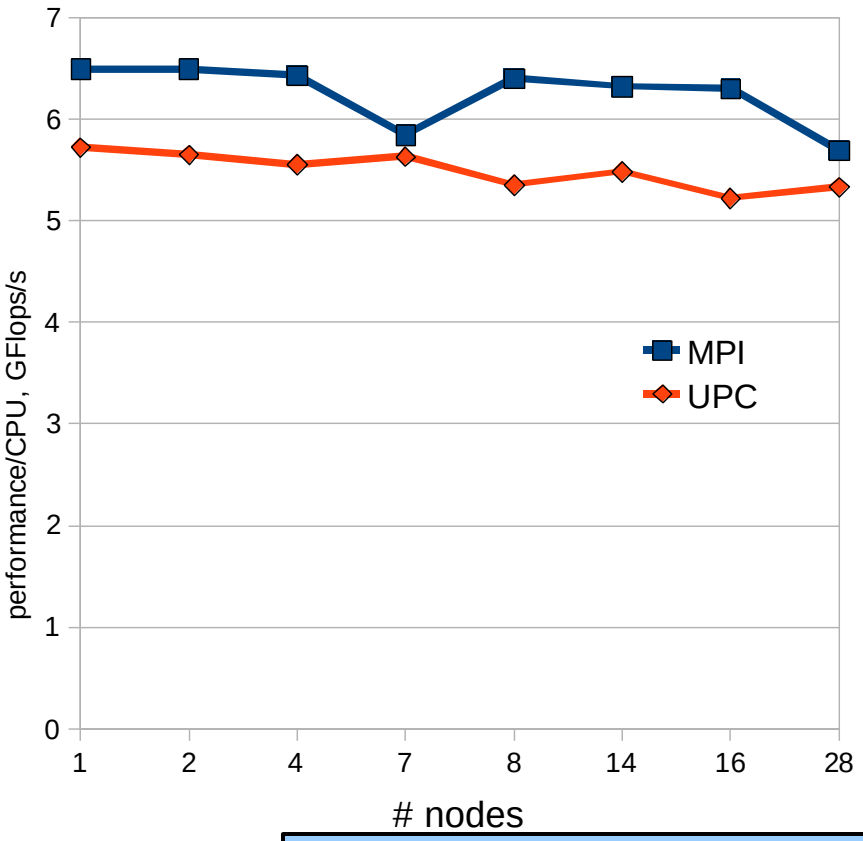
Memory Locality

- Shared memory physically located on the node where the thread is running has lower latency than memory located on another node (affinity concept)
- Compiler can exploit memory affinity information when thread mapping and cluster topology is known (static threads, num. nodes known)

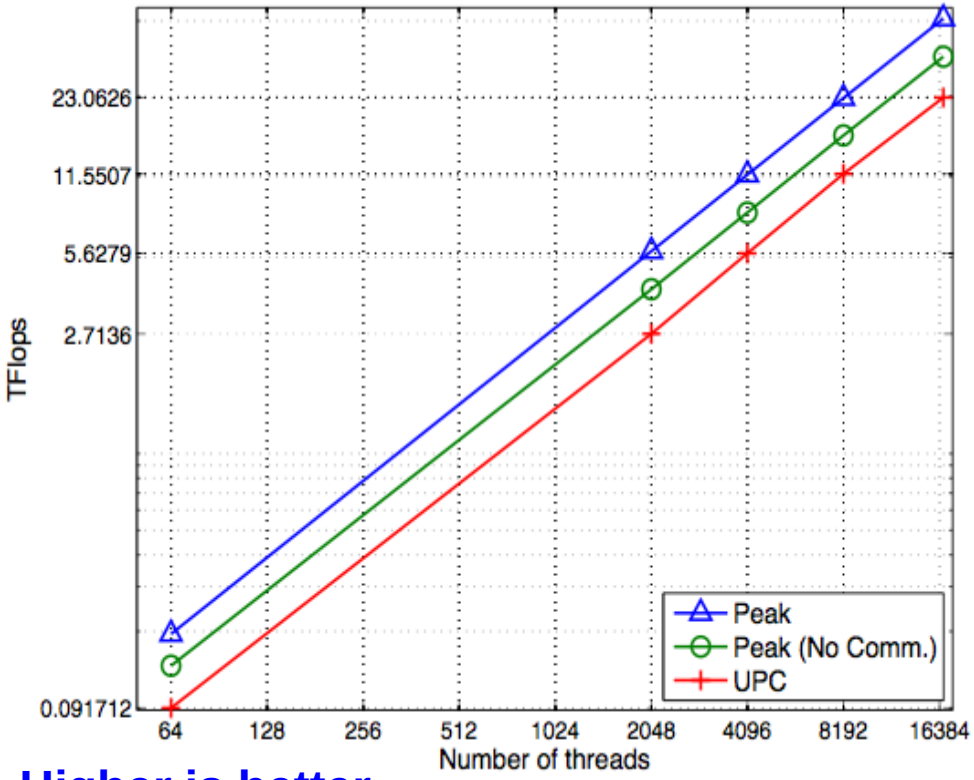
High Performance Linpack – MPI vs UPC

MPI vs UPC HPL

Power5 + HPS, 28 nodes, 16 processes/node



UPC HPL (Blue Gene/L, 16k nodes)



Higher is better

Performance: almost equivalent, lags **10%** behind MPI
Complexity: UPC code **1,430** lines, MPI code **30,744** lines

XL UPC Language Extensions

Flexible data layout: tiled arrays

example: **shared [2][2] int A[4][4];**

provides ability to call to efficient libraries, e.g., ESSL

Work in Progress

Mapping data to processors

Allows control of block placement on processor grid

Allows communication among rows, cols of processors

```
#pragma distributed A(2,2)
extern shared [2][2] double A[4][4];
```

Data driven team collectives

collective operation participants are a subset of the threads (team)

```
allreduce (team, &src, &dst, OP_ADD, dt, size);
```

defining teams based on data distribution fits better the in the programming model.

It can significantly reduce the amount of communication.

Example: add all elements in a row of processors

These language extensions make UPC more effective at optimizing a wider range of scientific applications:

NAS CG: performance of the UPC version within 10% of the MPI performance, but the UPC benchmark implementation uses 50% less code

HP Linpack: performance of the UPC version within 10% of the MPI performance (MPI code from http://icl.cs.utk.edu/hpcc), but with 20 times less code

Random Access: UPC version outperforms MPI version by ~15%

Coarray Fortran Compiler (CAF)

Programming Model: Single Program Multiple Data (SPMD)

- Fixed number of processes (images)

“Everything is local!” [Numerich]

- All data is local
- All computation is local

Explicit data partition with one-sided communication

- Remote data movement through codimensions

Programmer explicitly controls the synchronizations

- Good or bad?

CAF: Coarray syntax

CODIMENSION attribute

double precision, dimension(2,2), CODIMENSION[*] :: x

or simply use [ ] syntax

double precision :: x(2,2)[*]

a coarray can have a corank higher than 1

double precision :: A(100,100)[5,*]

from ANY single image, one can refer to the array x on image Q using [ ]

X(:,:)[Q]

e.g. Y(:,:) = X(:,:)[Q]

X(2,2)[Q] = Z

Coindexed objects

Normally the remote data

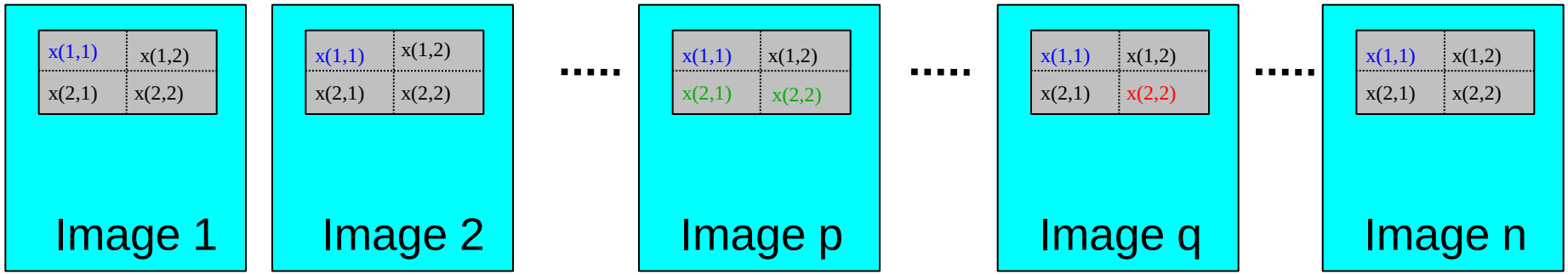
Without [ ] the data reference is local to the image

X(1,1) = X(2,2)[Q]

!LHS is local data; RHS is a coindexed object, likely a

!remote data

CAF: Coarray memory model



Logical view of coarray X(2,2)[*]

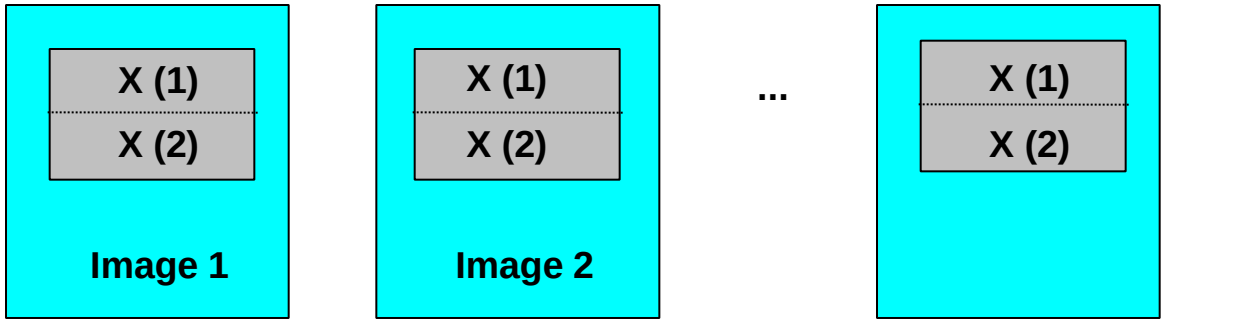
- A fixed number of images during execution
  - Each has a local array of shape (2 x 2)

examples of data access: local data and remote data

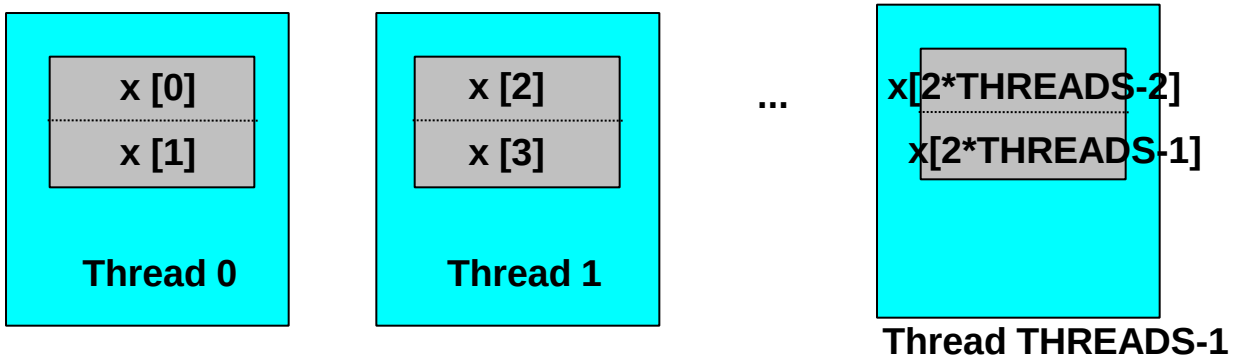
```
X(1,1)    =  X(2,2)[q] !assignment occurs on all images
if (this_image() == 1) X(2,2)[q] = SUM(X(2,:)[p])
                       !computation of SUM occurs on image 1
```

Comparison between CAF and UPC

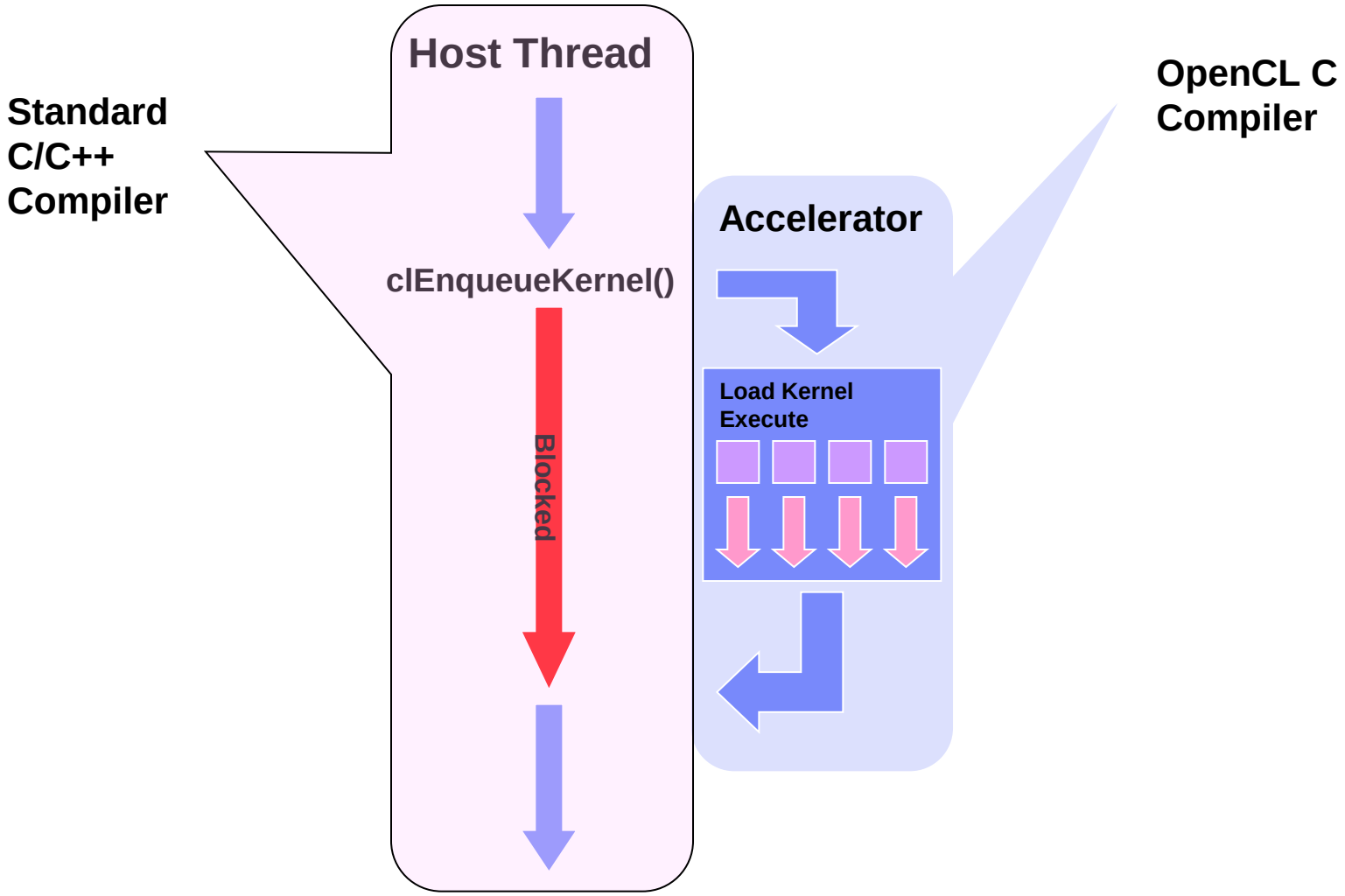
CAF : REAL :: X(2)[*]



UPC : shared [2] float x[2*THREADS]



OpenCL Compilers



OpenCL Compilers

Work in Progress

- Internal prototype on OpenCL compiler for CELL and POWER processors
- Some experience with implementing OpenMP and OpenCL on CELL
- Possible advantages of OpenCL versus OpenMP:

Architecture development provides the underlining environment
- Memory bandwidth, power consumption, simplified pipeline
- Cost effective GPU, accelerator

Massive data parallel programming
- Computation grid

Model remote memory
- Local memory: fast, no coherence
- Remote memory: global addressed, slow to access
- __global, __local

Builtin SIMD element
- Logical vector
- Data alignment considered

OpenCL - Transpose

Cell SDK – C with intrinsics

```
void transpose (vector float m[4])
{
  vector float abcd, efgh, ijkl, mnop;      /* input vectors */
  vector float aeim, bfjn, cgko, dhlp;      /* output vectors */
  vector float aibj, ckdl, emfn, gohp;      /* intermediate vectors */

  vector unsigned char shufflehi = VEC_LITERAL(vector unsigned char,
                                                0x00, 0x01, 0x02, 0x03
                                                0x10, 0x11, 0x12, 0x13,
                                                0x04, 0x05, 0x06, 0x07,
                                                0x14, 0x15, 0x16, 0x17);
  vector unsigned char shufflelo = VEC_LITERAL(vector unsigned char,
                                                0x08, 0x09, 0x0A, 0x0B,
                                                0x18, 0x19, 0x1A, 0x1B,
                                                0x0C, 0x0D, 0x0E, 0x0F,
                                                0x1C, 0x1D, 0x1E, 0x1F);
  abcd = m[0];
  efgh = m[1];
  ijkl = m[2];
  mnop = m[3];

  aibj = spu_shuffle(abcd, ijkl, shufflehi);
  ckdl = spu_shuffle(abcd, ijkl, shufflelo);
  emfn = spu_shuffle(efgh, mnop, shufflehi);
  gohp = spu_shuffle(efgh, mnop, shufflelo);

  aeim = spu_shuffle(aibj, emfn, shufflehi);
  bfjn = spu_shuffle(aibj, emfn, shufflelo);
  cgko = spu_shuffle(ckdl, gohp, shufflehi);
  dhlp = spu_shuffle(ckdl, gohp, shufflelo);

  m[0] = aeim;
  m[1] = bfjn;
  m[2] = cgko;
  m[3] = dhlp;
}
```

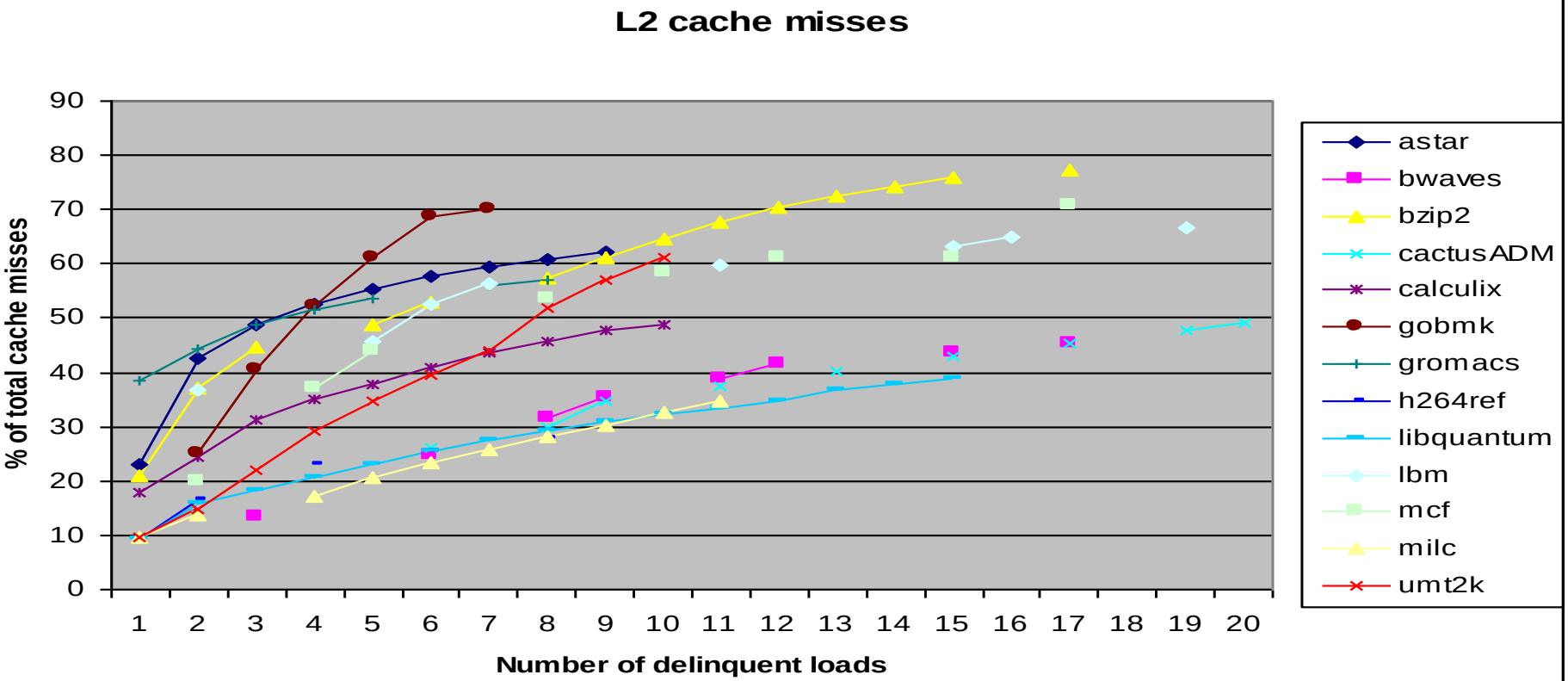
OpenCL C

```
void transpose (float4 m[4])
{
                float16 x = (float16) (m[0], m[1], m[2], m[3]);
                float16 t;

                t.even = x.lo;
                t.odd = x.hi;
                x.even = t.lo;
                x.odd = t.hi;

                m[0] = x.lo.lo;
                m[1] = x.lo.hi;
                m[2] = x.hi.lo;
                m[3] = x.hi.hi;
}
```

Delinquent Load Identification

Very small number of *delinquent* loads are responsible for the vast majority of cache misses

Delinquent load identification: User annotation / Static analysis
Dynamic profiling



Static Analysis for Delinquent Load Identification

Array recovery for pointer access (e.g. p++.)

Unique array identification (memory disambiguation)

Access pattern analysis for stream identification
- Stride one stream (unit stride access, a[i])
- Stride N stream (A[i][*][*], B[N*i]);
- Indexed stream(A[B[i]);
- Irregular access (p->next, p->right, p->left, …)

Reuse analysis

Dynamic Profiling for Delinquent Load Identification

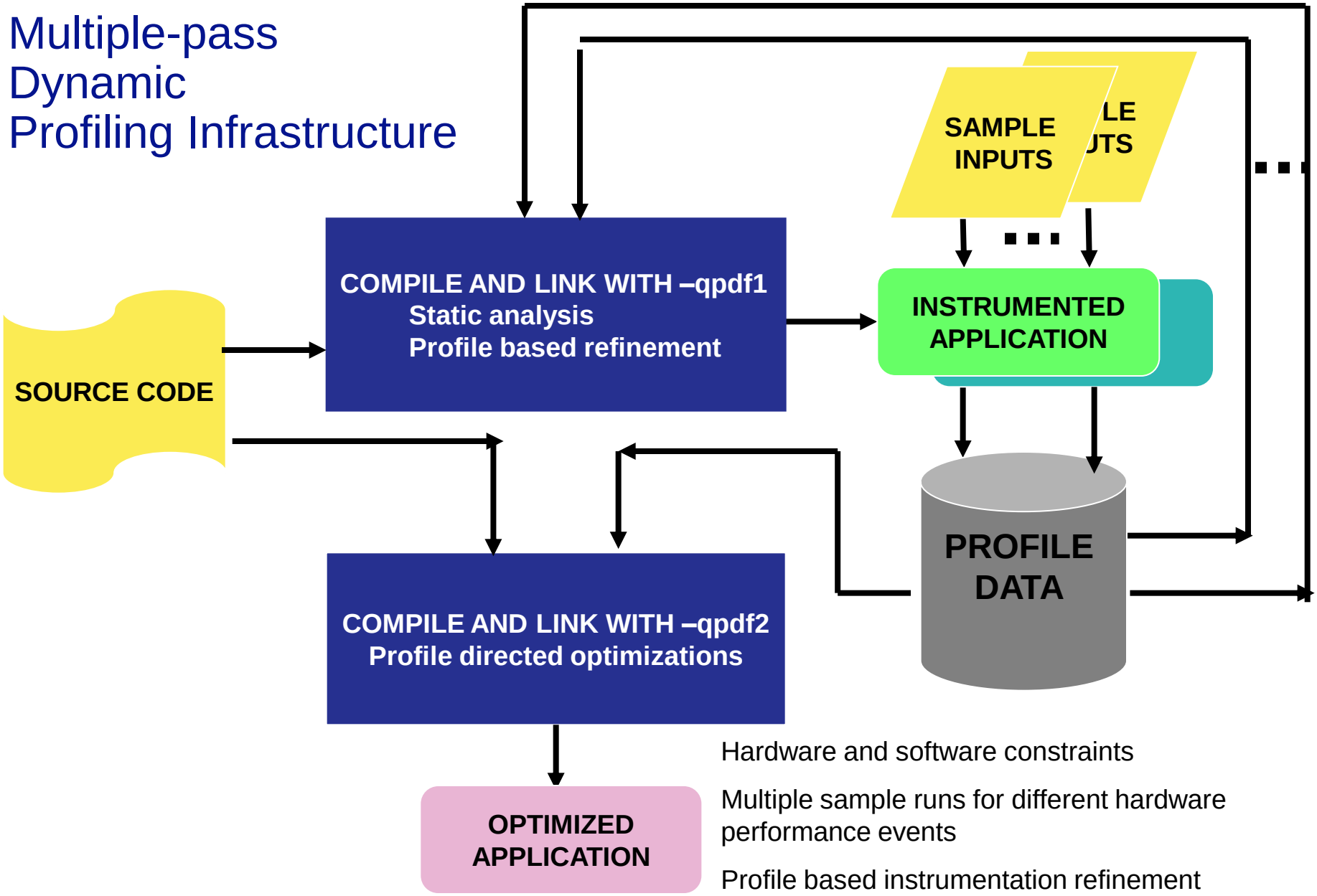
Coarse-Grain Cache Miss Profiling

- Identify the code regions which contain the memory operations which cause most cache misses
- Generate pmapi calls or the code to read performance counters directly

Fine-Grain Cache Miss Profiling

- Identify specific memory operations which cause cache misses
- Combine static analysis to instrument memory operations which most likely cause runtime cache misses
- Generate the code to read performance counter directly

Multiple-pass Dynamic Profiling Infrastructure



- Hardware and software constraints
- Multiple sample runs for different hardware performance events
- Profile based instrumentation refinement

Cache Miss Information

Delinquent load

Memory Reference	Cache Level	Cache Miss Count	Miss Rate	Line Number	Function	File
((char * *)((char *)@CIVINIT4 + @CIV4 * 60))->(*)node.node.basic_arc.rns38.->(*)arc.arc.flow.rns39.	2	45331	63	147	flow_cost	mcfutil.c
arc2->(*)arc.arc.flow.rns14.	2	885002	95	60	write_circulations	output.c
bla->(*)arc.arc.ident.rns28.	2	45274	43	123	primal_net_simplex	psimplex.c
@ICM2->(*)long.rns8.	2	40	71	127	primal_net_simpl	psimplex.c
temp.rnn4->(*)node.node.sibling.rns8.->(*)node.node.sibling_prev.rns9.	2	714	7	102	update_tree	treeup.c

Cache misses

Source code location

Delinquent Load Driven Optimization

Prefetch Effectiveness

- **Timeliness**
  - the placement of the prefetches such that the latency to memory is effectively hidden
- **Accuracy**
  - prefetching data which will actually be used by the program to avoid cache pollution
- **Overhead**
  - incurring the least amount of overhead incurred by the prefetch instructions

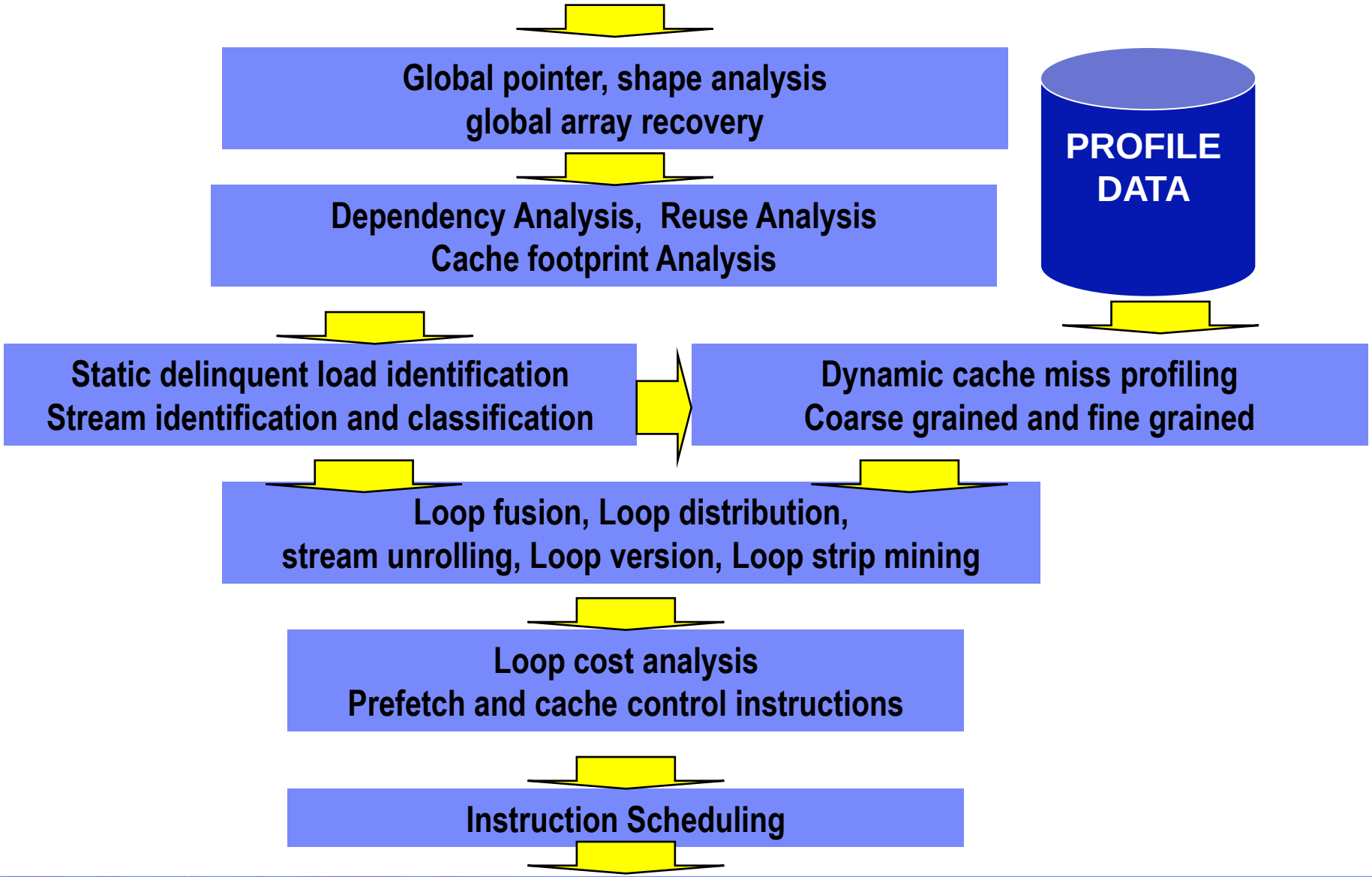
Data prefetch

- Software assisted data prefetch to reduce the overhead of hardware stream identification
- Prefetch depth control
- Prefetch instruction selection
- Prefetch engine control

Instruction Scheduling

- Instruction reordering to hide memory latency
- Prefetch instruction awareness scheduling

Delinquent Load Driven Optimizations



Assist Threads: Motivation and Goals

Motivation:

Significant amount of cache misses in HPC workloads

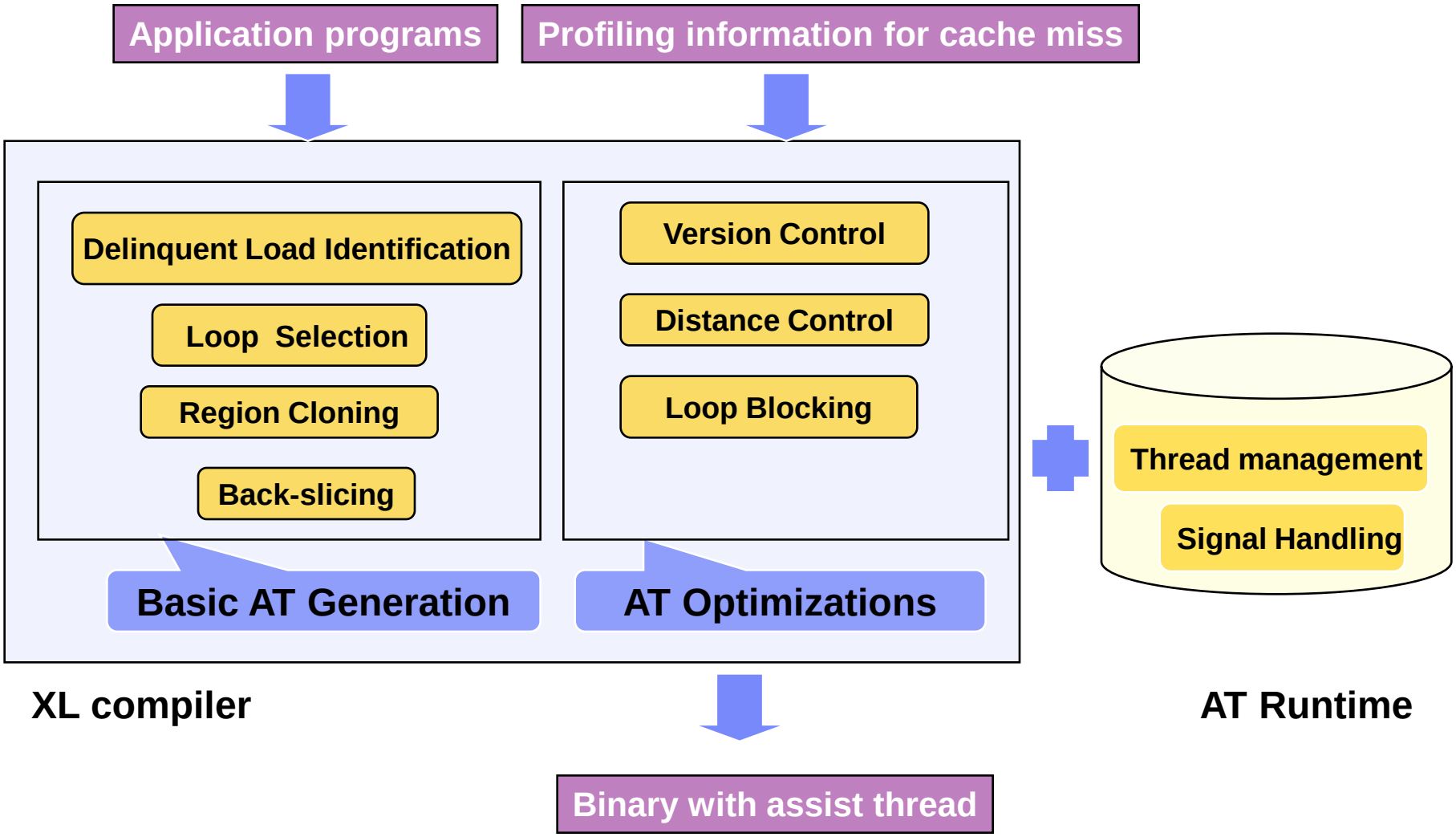
Availability of multi-core and multi-threading for CMP (Chip MultiProcessors) and SMT (Simultaneous MultiThreading) exploitation

Existing prefetch techniques have limitations:
- hardware prefetch - irregular data access patterns
- software prefetch – prefetch overhead on program execution

Goals:

Deploy the available multiple SMT threads and cores to increase single thread and multiple thread performance

Assist Threads: Compiler Infrastructure



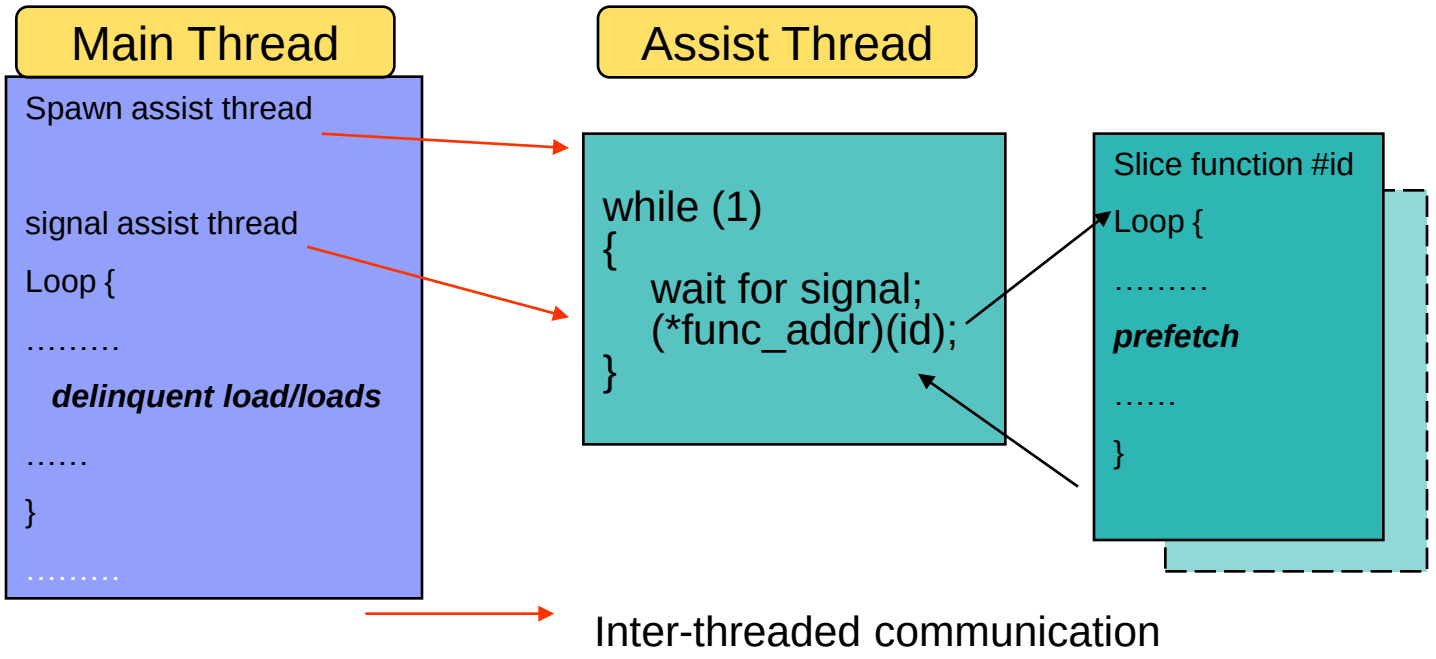
Assist Threads: Code Generation

Delinquent load identification (common to Memory Hierarchy Optimizations.)

Back-slicing for address computation and prefetch insertion

Spawn assist threads once in MT (Main Thread)

Notify AT (Assist Thread) to execute the slice function for delinquent loads



Example of Assist Threads Code Transformation

Original Code

```
// A and x are globals
y = func1();
…
i = func2();
// start of back-slice
while (i < condition) {
                …
                x = operations;
                …
                // delinquent load
                =…+A[i] ;
                …
                i += y;
}
// end of back-slice
```

Main Thread (MT) Code

```
y = func1();
…
i = func2();
// start pre-fetching in assist thread
func_addr = &slice_func_1;
signal assist thread;
// start of back-slice
while (i < condition) {
                …
                x = operations
                …
                =…+ A[i] ;
                …
                i += y;
}
// end of back-slice
```

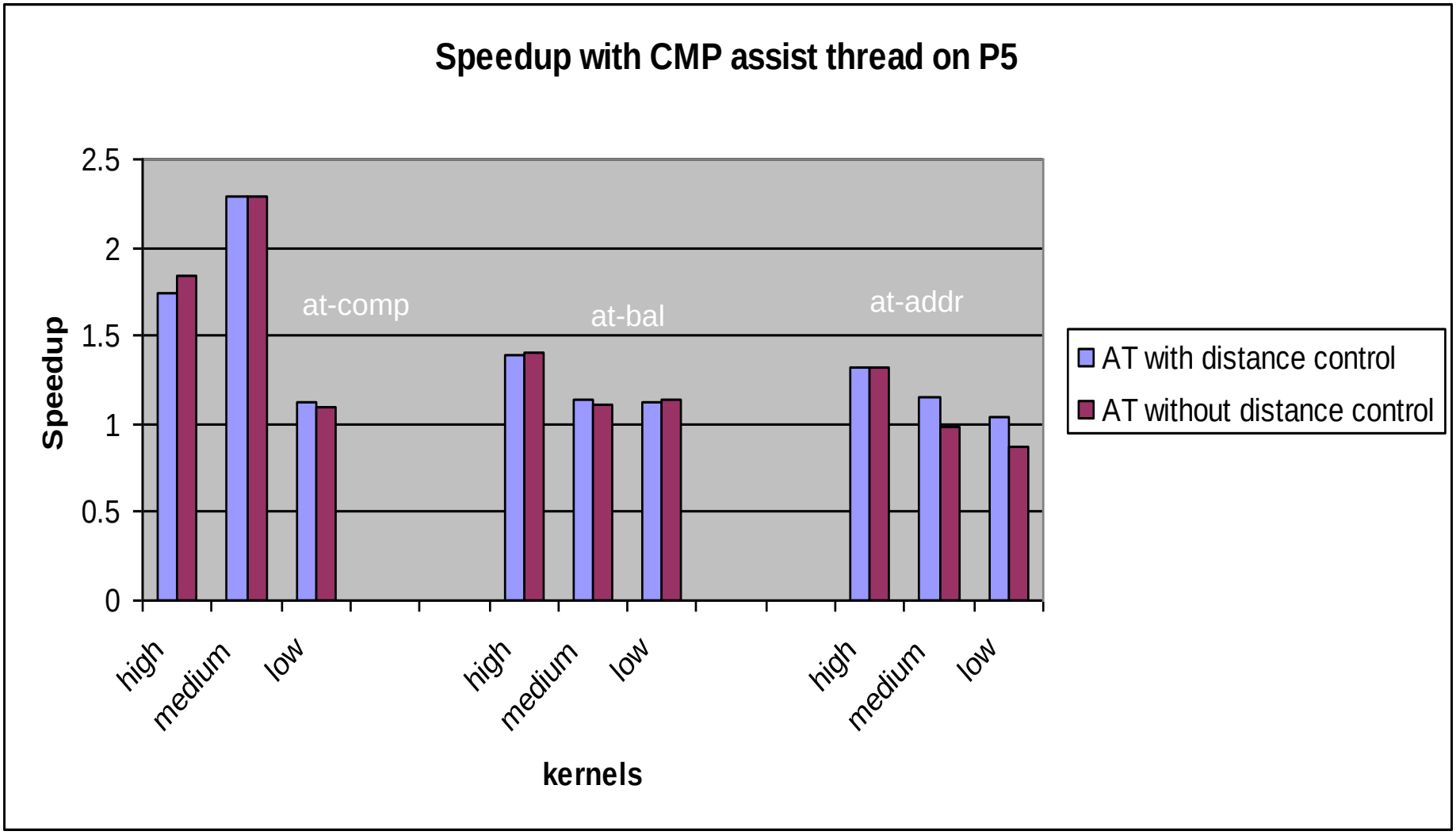
Assist Thread (AT) Slice Function

```
void slice_func_1(int thd_id) {
…
  int y_local = y;
  int i_local = i;
…
while (i_local < condition) {
                // pre-fetch request
                __dcbt( &A[i_local] );
                i_local += y_local;
  }
}
```

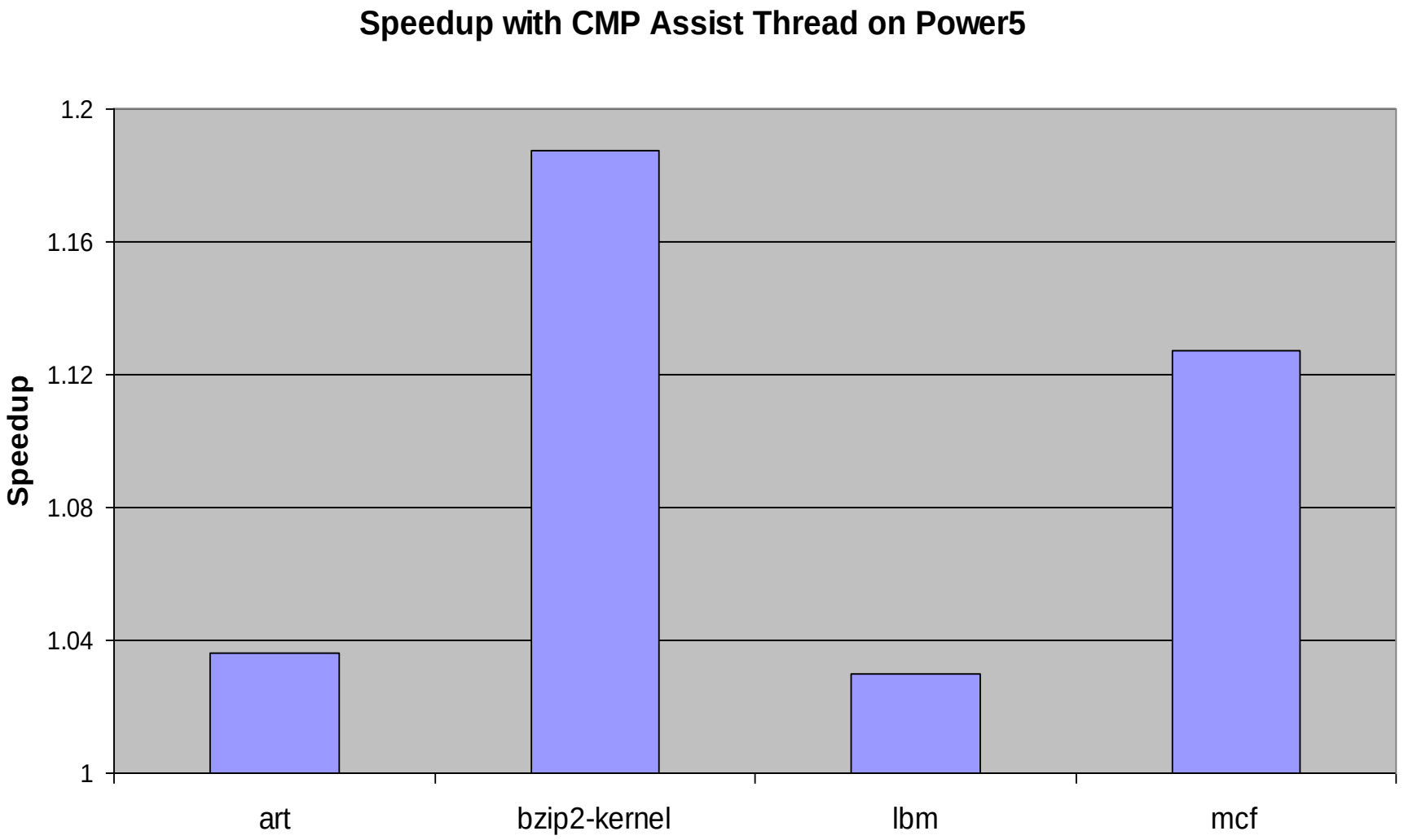
Assist Threads: Kernels Description

- Synthesis test cases to show the performance of assist thread on
  - Different function unit usage
  - Different cache miss rate
- Operations in main thread can be grouped into:
  - ADDR: operations needed by AT to calculate the addresses for prefetch
  - COMP: the rest operations (computation) done by MT but not AT
- Different ratio between ADDR and COMP
  - at-comp: much more operations in COMP
  - at-addr: much more operations in ADDR
  - at-bal: ADDR and COMP are roughly balanced
- Different cache miss rate for delinquent loads ONLY
  - High: miss rate: ~90%
  - Medium: miss rate: ~40%
  - Low: miss rate: ~20%

Assist Threads: CMP Performance on Power5



Assist Threads: Speedup of Benchmarks



Assist Threads: Summary

Compiler exploitation of assist threads to demonstrate the performance improvement

- Delinquent load infrastructure integration
- Profitability analysis for code region selection
- Outlining and backward slicing
- Assist thread optimizations: distance control, loop blocking
- Speculation handling

User annotated and automatic assist thread generation will be implemented under the –qprefetch=AssistThread compiler option

All information subject to change without notice

Compiler Transformation Report Infrastructure

The compiler transformation infrastructure has been designed and implemented.

- The infrastructure allows the compiler to generate XML reports detailing the performed optimizations and missed optimization opportunities. The content of the XML reports has been enhanced to report on optimizations across all phases of the compilation.

XML reports are transformable

- XML can be translated into HTML with the style sheet into a human readable format
- XML is an easily consumable format which can be used by tooling with the XML schema

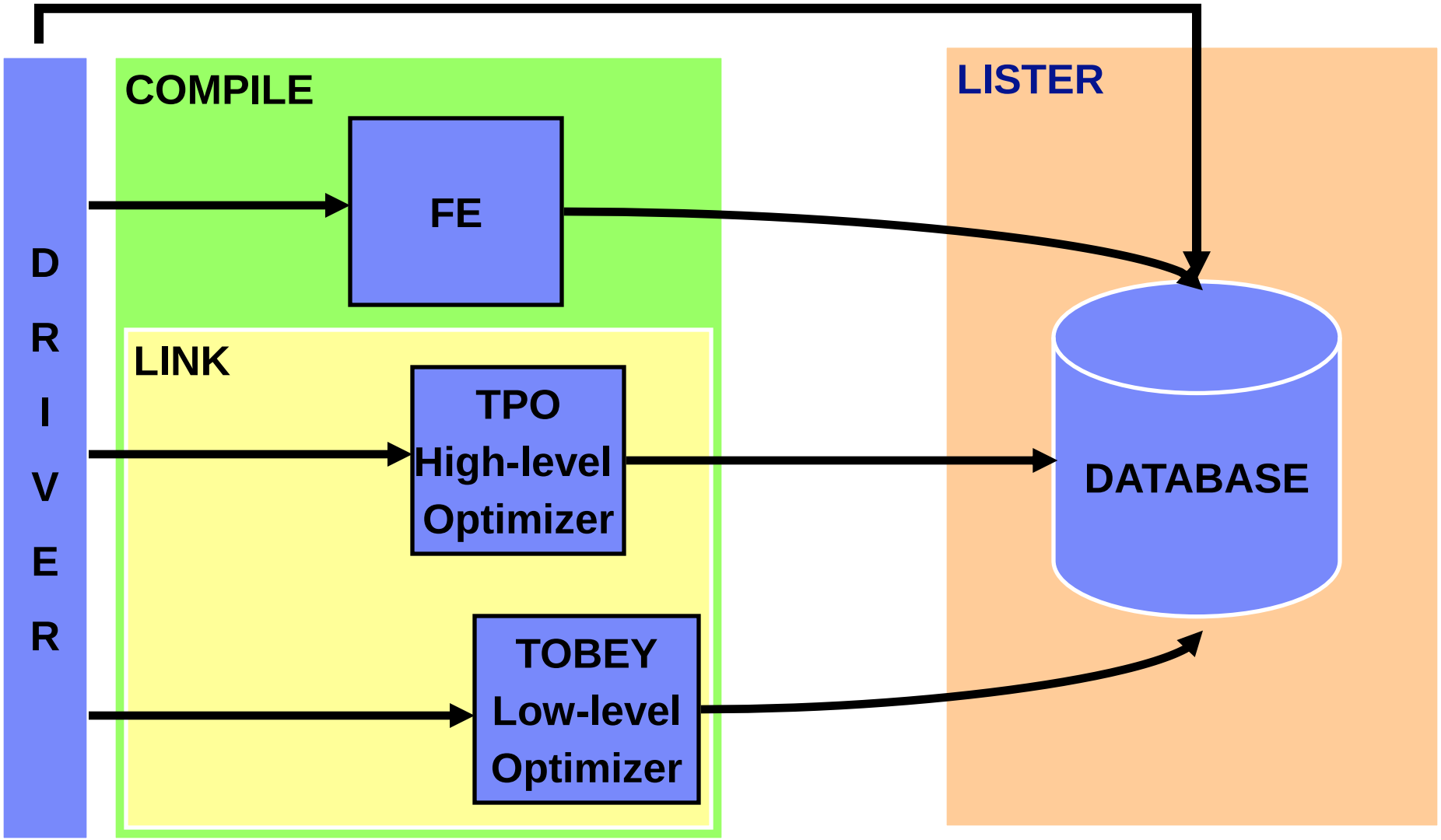
Compiler Transformation Reports enhance the programming productivity

- Increase productivity of manual code tuners by providing compiler information;
  - that would otherwise not be available to them
  - only available through more time consuming analysis by higher skilled users
    - (e.g. assembler listings)
- Increase productivity with integrated performance tools that do automatic or user assisted performance tuning (e.g. Hot spot identification by tools, combined with compiler optimization information on the hot spot)

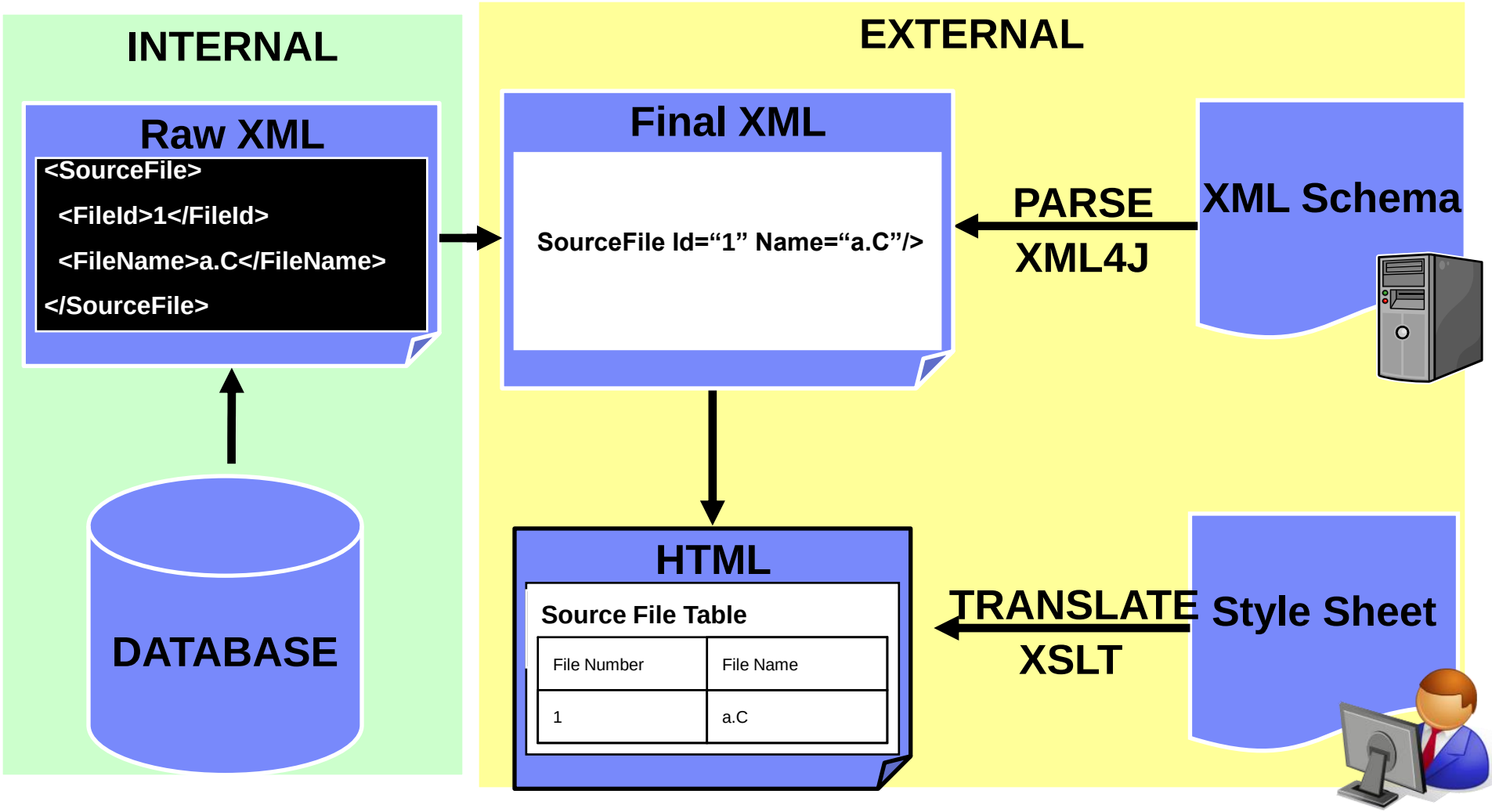
Compiler Transformation Report Types

- Inlining
  - Summarizes all successful inlines and all failing user inlines
  - Represents inlining from all phases of the compilation (Front End, High Level Optimizer, and Low Level Optimizer)
- Transformations
  - Shows a list of successful transforms
  - Shows a list of unsuccessful transforms with reasons for failure
  - Represents transformations preformed by the High Level Optimizer during both compile and link phases, including:
    - Loop transformations
    - Parallelizatio transformations
    - Vectorization/SIMDization transformations
    - Data prefetch
- Data Reorganization
  - Reports a summary of variable data reorganized by the High Level Optimizer

Compiler Transformation Report Infrastructure



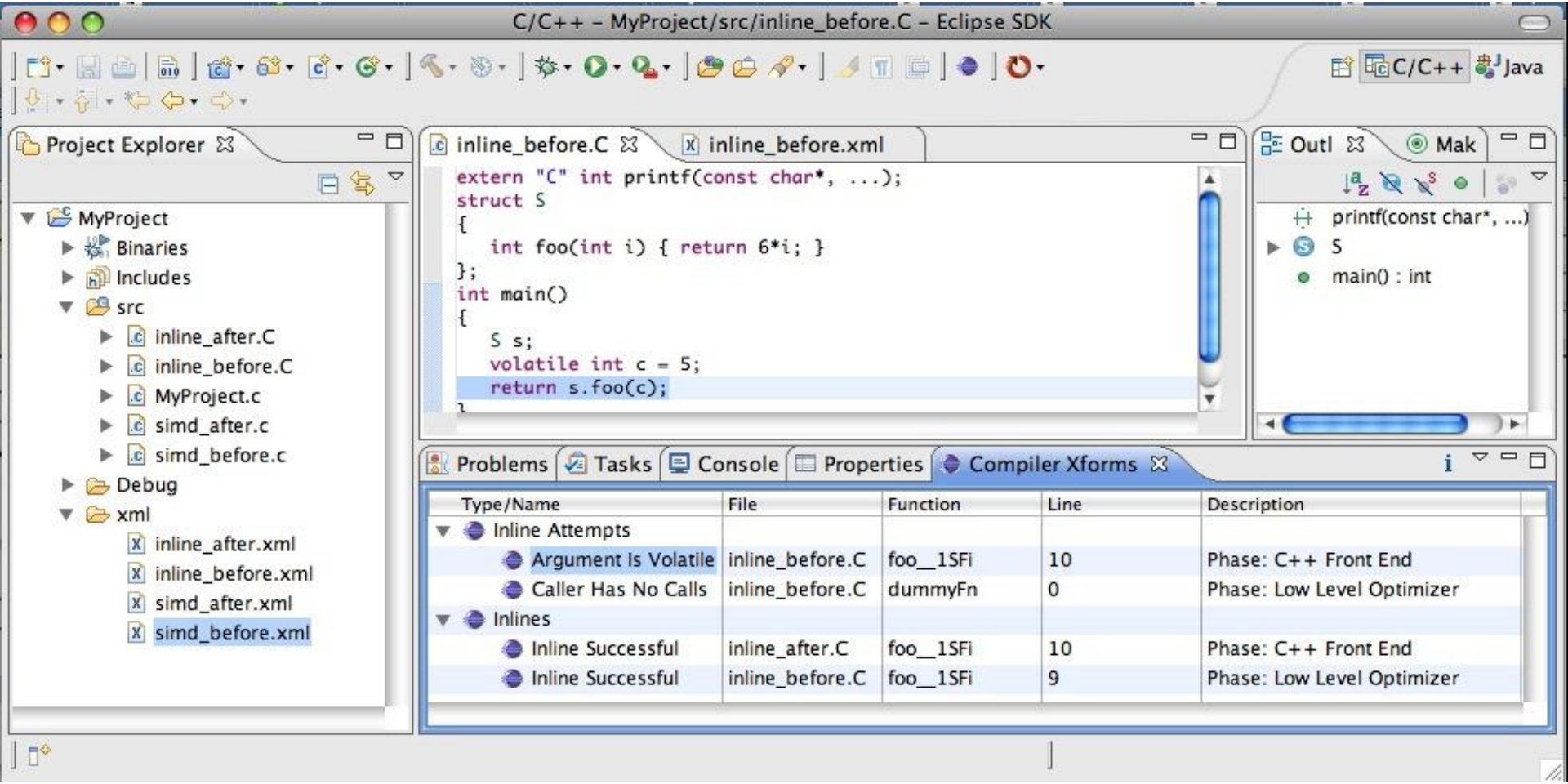
Compiler Transformation Report Infrastructure



Eclipse View

Transformation report information in Eclipse:

- **More easily readable**
- **Easy navigation to source code locations**



Automatic Parallelization

Work in Progress

Enablement:

- Removing dependencies is key to enable more parallelization
- Array Privatization
- Runtime dependence test
- Interprocedural array section analysis to parallelize loops with calls
- Multi dimensional array reductions (common in scientific codes)

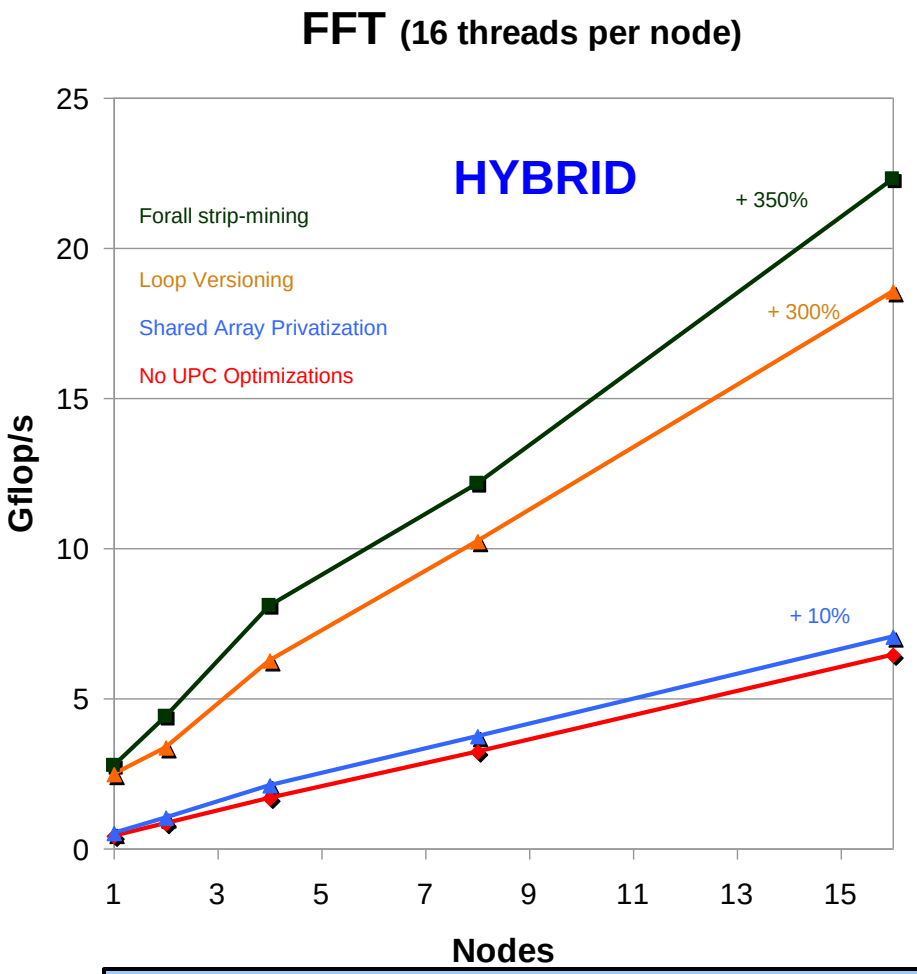
Cost Analysis:

- Parallelization is not always profitable
- Both compile-time and runtime analysis is done
- Runtime profiling of sequential/parallel code

Gaining valuable experience parallelizing SPECFP

BACKUP SLIDES

FFT - Compiler Optimizations




```
typedef struct { double re, im; }  complex_t;
typedef shared [BF] complex_t   ComplexArray_t [N*N];

void transpose2 (ComplexArray_t A, ComplexArray_t B) {
   // tile exchange (communication)
   upc_barrier;
   // local transpose
   upc_forall(i = 0; i < N; i += bsize; &B[i*N])
     for (j = 0; j < N; j += bsize)
       for (ArrayIndex_t k = 0; k < bsize - 1; k++)
         for (ArrayIndex_t l = k + 1; l < bsize; l++) {
           complex_t c = B[(i+k)*N+(j+l)];
           B[(i+k)*N+(j+l)] = B[(i+l)*N+(j+k)];
           B[(i+l)*N+(j+k)] = c;
         }
```

Privatized

1. version the upc_forall, this allows the analysis to reason about the locality of shared accesses done through pointers-to-shared
2. accesses in the local transpose are then recognized as local and privatized.

CAF: Standardization status

Coarray is in base language of Fortran 2008

- Standard to be published in 2010
- Fortran to be the first general purpose language to support parallel programming

The coarray TR (future coarray features)

- TEAM and collective subroutines
- More synchronization primitives
  - notify / query (point – to – point)
- Parallel IO: multiple images on same file

Coarrays and MPI

Early experience demonstrated coarrays and MPI can coexist in the same application

Migration from MPI to coarray has shown some success

- Major obstacle: CAF is not widely available

Fortran J3 committee willing to work with MPI forum

- Two issues Fortran committee is currently working on to support:
  - C interop with void *

    void * buf; (C)

    TYPE(*), dimension(...) :: buf (Fortran)
  - MPI nonblocking calls: **MPI_ISEND, MPI_IRECV and MPI_WAIT**

Example: comparing CAF to MPI

MPI:

```
if (master) then
  r(1) = reynolds
  ...
  r(18) = viscosity
  call mpi_bcast(r,18,real_mp_type,
                 masterid,
                 MPI_comm_world,
                 ierr)
else
  call mpi_bcast(r, 18,
                 real_mp_type,
                 masterid,
                 MPI_comm_world,
                 ierr)
  reynolds = r(1)
   ...
   viscosity = r(18)
endif
```

(Ashby and Reid, 2008)

CAF:

```
sync all
if (master) then
  do i=1, num_images()-1
    reynolds[i] = reynolds
    ...
    viscosity[i] = viscosity
    end do
end if
sync all
```

Or simply:

```
sync all
reynolds = reynolds[masterid]
...
viscosity = viscosity[masterid]
```

XL Fortran Roadmap: Strategic Priorities

Superior Customer Service

Continue to work closely with key ISVs and customers in scientific and technical computing industries

Compliance to Language Standards and Industry Specifications

OpenMP API V2.5 (Full) and OpenMP API V3.0 (Partial)

Fortran 77, 90 and 95 standards

Fortran 2003 Standard

Exploitation of Hardware

Committed to maximum performance on POWER4, PPC970, POWER5, POWER6, PPC440, PPC450, CELL and successors

Continue to work very closely with processor design teams

XL C/C++ Roadmap: Strategic Priorities

Superior Customer Service

Compliance to Language Standards and Industry Specifications

- ANSI / ISO C and C++ Standards
- OpenMP API V3.0

Exploitation of Hardware

- Committed to maximum performance on POWER4, PPC970, POWER5, PPC440, POWER6, PPC450, CELL and successors
- Continue to work very closely with processor design teams

Exploitation of OS and Middleware

- Synergies with operating system and middleware ISVs (performance, specialized function)
- Committed to AIX Linux affinity strategy and to Linux on pSeries

Reduced Emphasis on Proprietary Tooling

- Affinity with GNU toolchain

Documentation

An information center containing the documentation for the **XL Fortran V12.1** and **XL C/C++ V10.1** versions of the AIX compilers is available at:
http://publib.boulder.ibm.com/infocenter/comphelp/v101v121/index.jsp

An information center containing the documentation for the **XL Fortran V11.1** and **XL C/C++ V9.0** versions of the AIX compilers is available at:
http://publib.boulder.ibm.com/infocenter/comphelp/v9v111/index.jsp

Optimization and Programming Guide for XLF V12.1 is now available online at:
http://publib.boulder.ibm.com/infocenter/comphelp/v101v121/index.jsp

New whitepaper “Overview of the IBM XL C/C++ and XL Fortran Compiler Family” available at: **http://www.ibm.com/support/docview.wss?uid=swg27005175**

This information center contains all the html documentation shipped with the compilers. It is completely searchable.

Please send any comments or suggestions on this information center or about the existing C, C++ or Fortran documentation shipped with the products to **compinfo@ca.ibm.com.**

Power Systems Compiler Products: Latest Versions

All POWER4, POWER5, POWER6 and PPC970 enabled

- XL C/C++ for AIX, V10.1 (July 2008)
- XL Fortran for AIX, V12.1 (July 2008)
- XL C/C++ for Linux, V10.1 (September 2008)
- XL Fortran for Linux, V12.1 (September 2008)

Blue Gene (BG/L and BG/P) enabled

- XL C/C++ Advanced Edition for BG/L, V9.0
- XL Fortran Advanced Edition for BG/L, V11.1
- XL C/C++ Advanced Edition for BG/P, V9.0
- XL Fortran Advanced Edition for BG/P, V11.1

Cell/B.E. cross compiler products:

- XL C/C++ for Multicore Acceleration for Linux on Power Systems, V10.1 (4Q2008)
- XL C/C++ for Multicore Acceleration for Linux on x86 Systems, V10.1 (4Q2008)
- XL Fortran for Multicore Acceleration for Linux on System p, V11.1

Power Systems Compilers : Latest Versions

Technology Preview currently available from alphaWorks

XL UPC language support on AIX and Linux

Download: **http://www.alphaworks.ibm.com/tech/upccompiler**

XL C/C++ for Transactional Memory for AIX

Download: **http://www.alphaworks.ibm.com/tech/xlcstm**

CDT for AIX

Download: http**://www.alphaworks.ibm.com/tech/cremoteide**

IBM Debugger for AIX (with Fortran support)

Download:

https://www.ibm.com/services/forms/preLogin.do?source=swerpsw

History Of Compiler Improvement On Power4

Compilers	2001 V5/V7.1.1	2002 V6/V8.1	2003 V6/V8.1.1	2004 V7/V9.1	2005 V8/V10.1	Compound Over 4 Years	CAGR Rate
SpecINT	Baseline	21%	0%	3%	7%	34%	7.6%
SpecFLOAT	Baseline	12%	5%	18%	5%	46%	9.9%

Note: SPEC2000 base options improvements from www.spec.org

History Of Compiler Improvement On Power5

Compilers	2004 V7/V9.1	2005 V8/V10.1	2007 V9/V11.1	Compound Over 3 Years
SpecINT	Baseline	4.3%	6.4%	11%
SpecFLOAT	Baseline	5.4%	1.8%	7.3%

Note: SPEC2000 base options improvements from www.spec.org

A Unified Simdization Framework