X10: Programming for Hierarchical Parallelism and NonUniform Data Access

Kemal Ebcioglu

Vivek Sarkar

Vijay Saraswat

IBM T.J. Watson Research Center

vsarkar@us.ibm.com

LaR 2004 Workshop

OOPSLA 2004

This work has been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004.

Acknowledgments: PERCS team

- **IBM PERCS Team members**
  - IBM Research
  - IBM Systems & Technology Group
  - IBM Software Group
  - PI: Mootaz Elnozahy

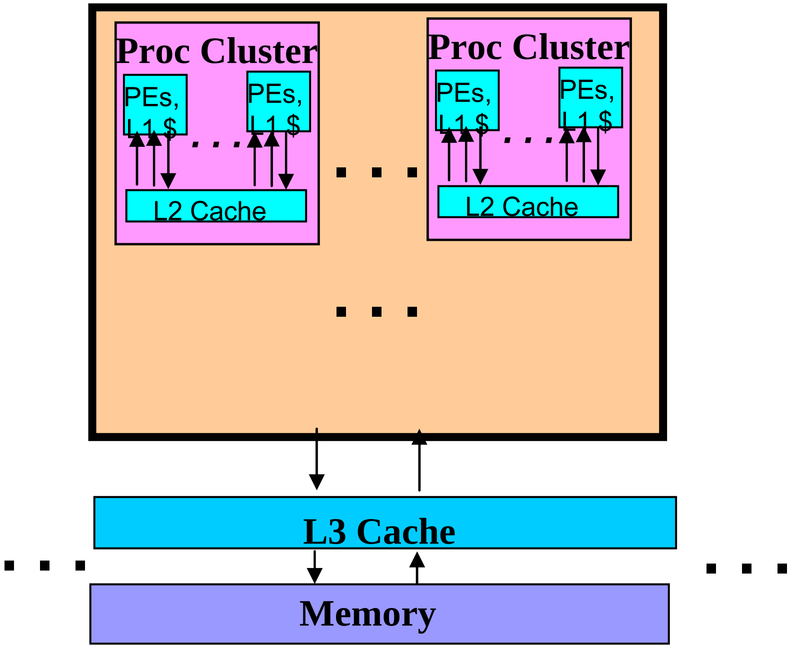
- **University partners:**
  - Cornell
  - LANL
  - MIT
  - Purdue University
  - RPI
  - UC Berkeley
  - U. Delaware
  - U. Illinois
  - U. New Mexico
  - U. Pittsburgh
  - UT Austin
  - Vanderbilt University

- **X10 core team**
  - Philippe Charles
  - Kemal Ebcioglu
  - Patrick Gallop
  - Christian Grothoff (Purdue)
  - Christoph von Praun
  - Vijay Saraswat
  - Vivek Sarkar

- **Additional contributors to X10 design & implementation ideas:**
  - David Bacon
  - Bob Blainey
  - Perry Cheng
  - Julian Dolby
  - Guang Gao (U Delaware)
  - Allan Kielstra
  - Robert O'Callahan
  - Filip Pizlo (Purdue)
  - V.T.Rajan
  - Lawrence Rauchwerger (Texas A&M)
  - Mandana Vaziri
  - Jan Vitek (Purdue)

Performance and Productivity Challenges facing Future Scalable Systems

1) **Memory wall:** Severe *non-uniformities* in bandwidth & latency in memory hierarchy

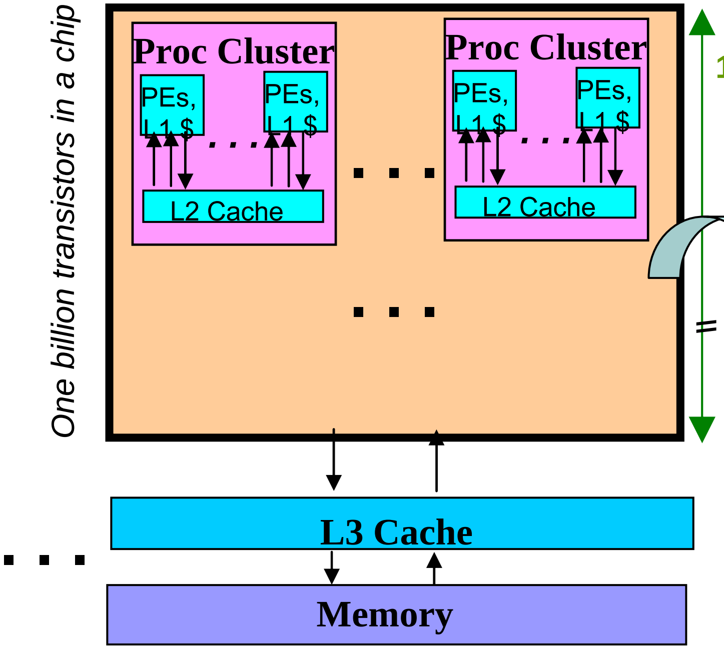


2) **Frequency wall:** Multiple layers of *hierarchical heterogeneous parallelism* to compensate for slowdown in frequency scaling

Clusters (scale-out)
SMP
Multiple cores on a chip
Coprocessors (SPUs)
SMTs
SIMD
ILP

3) **Scalability wall:** Software will need to deliver ~ 10^5-way *parallelism* to utilize peta-scale parallel systems

High Complexity of HPC Systems Limits HPC Application Development Productivity



1995: entire chip can be accessed in 1 cycle

2010: only small fraction of chip can be accessed in 1 cycle

Major sources of complexity for application developer:
1) Severe non-uniformities in data accesses
2) Applications must exhibit large degrees of parallelism (up to ~ 10^5 threads)

Complexity leads to increases in all phases of HPC Software Lifecycle related to parallel code

- Requirements
- Input Data
- Algorithm Development
- Written Specification
- Source Code
- Parallel Specification
- Development of Parallel Source Code --- Design, Code, Test, Port, Scale, Optimize
- Production Runs of Parallel Code
- Maintenance and Porting of Parallel Code

HPC Software Lifecycle

Impact of Programming Model on Productivity

1. **Safety** – how much of the burden of ensuring absence of errors falls on the user? e.g., Type errors, Initialization errors, Memory errors, Concurrency errors, Consistency errors, …
2. **Portability** – how much effort is required to move the application across multiple platforms and multiple system generations?
3. **Performance** --- how much of the burden of managing and tuning program resources falls on the user?
4. **Integration** --- to what extent can the programming model reuse existing Languages, Environment, Libraries, and Tools?

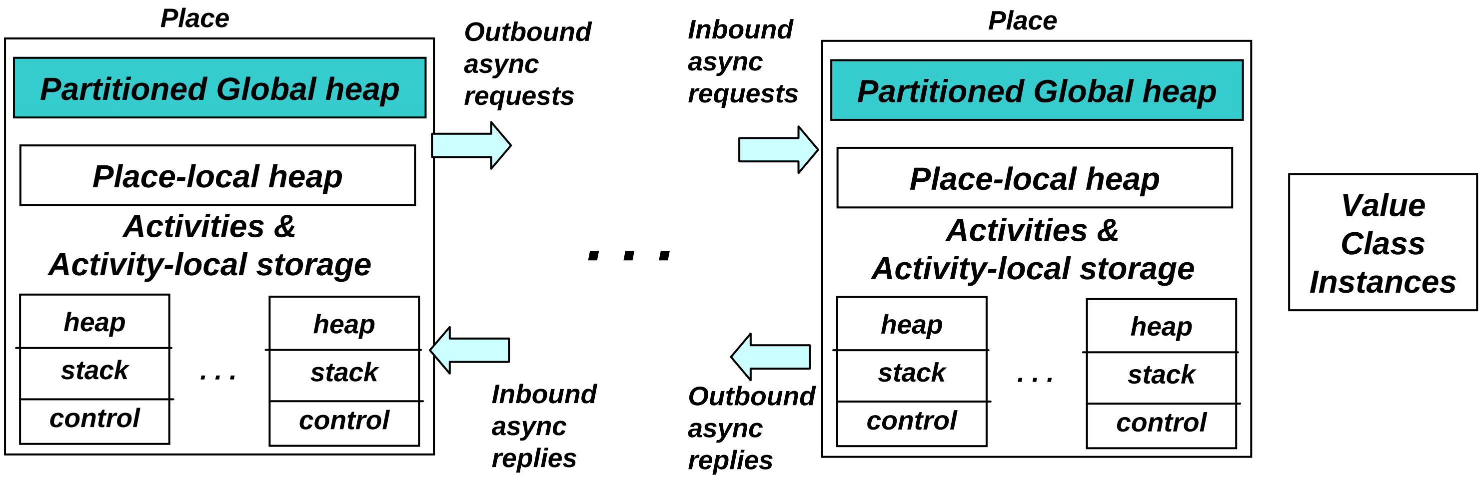
Impact of Programming Model on Compiler-Driven Performance

- **MPI: Local memories + message-passing**
  - **Parallelism, locality, and “global view” are completely managed by programmer**
  - **Communication, synchronization, consistency operations specified at low level of abstraction**
  - **➔ Limited *opportunities* for compiler optimizations**
- **Java threads, OpenMP: shared-memory parallel programming model**
  - **Uniform symmetric view of all shared data**
  - **Non-transparent performance --- programmer cannot manage data locality and thread affinity at different hierarchy levels (cluster, SMT, …)**
  - **➔ Limited *effectiveness* of compiler optimizations**
- **HPF, UPC: partitioned global address space + SPMD execution model**
  - **User specifies data distribution & parallelism, compiler generates communications using owner-computes rule**
  - **Large overheads in accessing shared data; compiler optimizations can help applications with simple data access patterns**
  - **➔ Limited *applicability* of compiler optimizations**

X10 Design Guidelines: Design for Productivity & Compiler/Runtime-driven Performance

- **Start with state-of-the-art OO language primitives as foundation**
  - No gratuitous changes
  - Build on existing skills
- **Raise level of abstraction for constructs that should be amenable to optimized implementation**
  - Monitors → atomic sections
  - Threads, DMA → async activities
  - Barriers → clocks
- **Introduce new constructs to model hierarchical parallelism and non-uniform data access**
  - Places
  - Distributions
- **Support common parallel programming idioms**
  - Data parallelism
  - Control parallelism
  - Divide-and-conquer
  - Producer-consumer / streaming
  - Message-passing
- **Ensure that every program has a well-defined semantics**
  - Independent of implementation
  - Simple concurrency model & memory model
- **Defer fault tolerance and reliability issues to lower levels of system**
  - Assume tightly-coupled system with dedicated interconnect

Logical View of X10 Programming Model



Place

Partitioned Global heap

Place-local heap

Activities &
Activity-local storage

heap
stack
control

. . .

heap
stack
control

Outbound
async
requests

Inbound
async
replies

. . .

Inbound
async
requests

Outbound
async
replies

Place

Partitioned Global heap

Place-local heap

Activities &
Activity-local storage

heap
stack
control

. . .

heap
stack
control

Value
Class
Instances

Granularity of place can range from single h/w thread to an entire scale-up system

- ***Place*** **= collection of resident activities and data**
  - **Maps to a data-coherent unit in a large scale system**
- **Four storage classes:**
  - **Partitioned global**
  - **Place-local**
  - **Activity-local**
  - **Value class instances**
    - **Can be copied/migrated freely**

- **Activities can be created by**
  - ***async statements*** **(one-way msgs)**
  - ***future expressions***
  - ***foreach & ateach constructs***
- **Activities are coordinated by**
  - ***Atomic sections***
    - ***Current restriction: all data accesses in an atomic section must be place-local***
  - ***Atomic locations***
  - ***Clocks*** **(generalization of barriers)**
  - ***Force*** **(for result of future)**

X10 Type System: Additional Features

- Unified type system
  - All data items are objects
- Value classes and clocked final
  - Immutable --- no updatable fields
- Type parameters
  - Places, distributions,
- Nullable
  - All types are non-null by default, need to explicitly declare a variable as nullable
  - For any type T, the type ?T (read: “nullable T”) contains all the values of type T, and a special null value, unless T already contains null.
- Support for both rectangular multidimensional arrays (matrices) and nested arrays
- . . .

X10 Runtime design issues

- Places
  - Typically, map one place per SMP node
  - Scenarios where multiple places/node could be useful
    - Virtual partitions
    - Coprocessors w/ DMA
    - Hierarchical places
- Local async/future operations
  - Similar to lightweight threads
- Remote async/future operations
  - Similar to active messages
  - Runtime system needs to marshall/unmarshall parameters and return values
- Possible implementation strategies for atomic sections
  - Only execute one atomic section at a time in a place
  - Analyzable atomic setcions
  - Transactional semantics

X10, in comparison with Java…

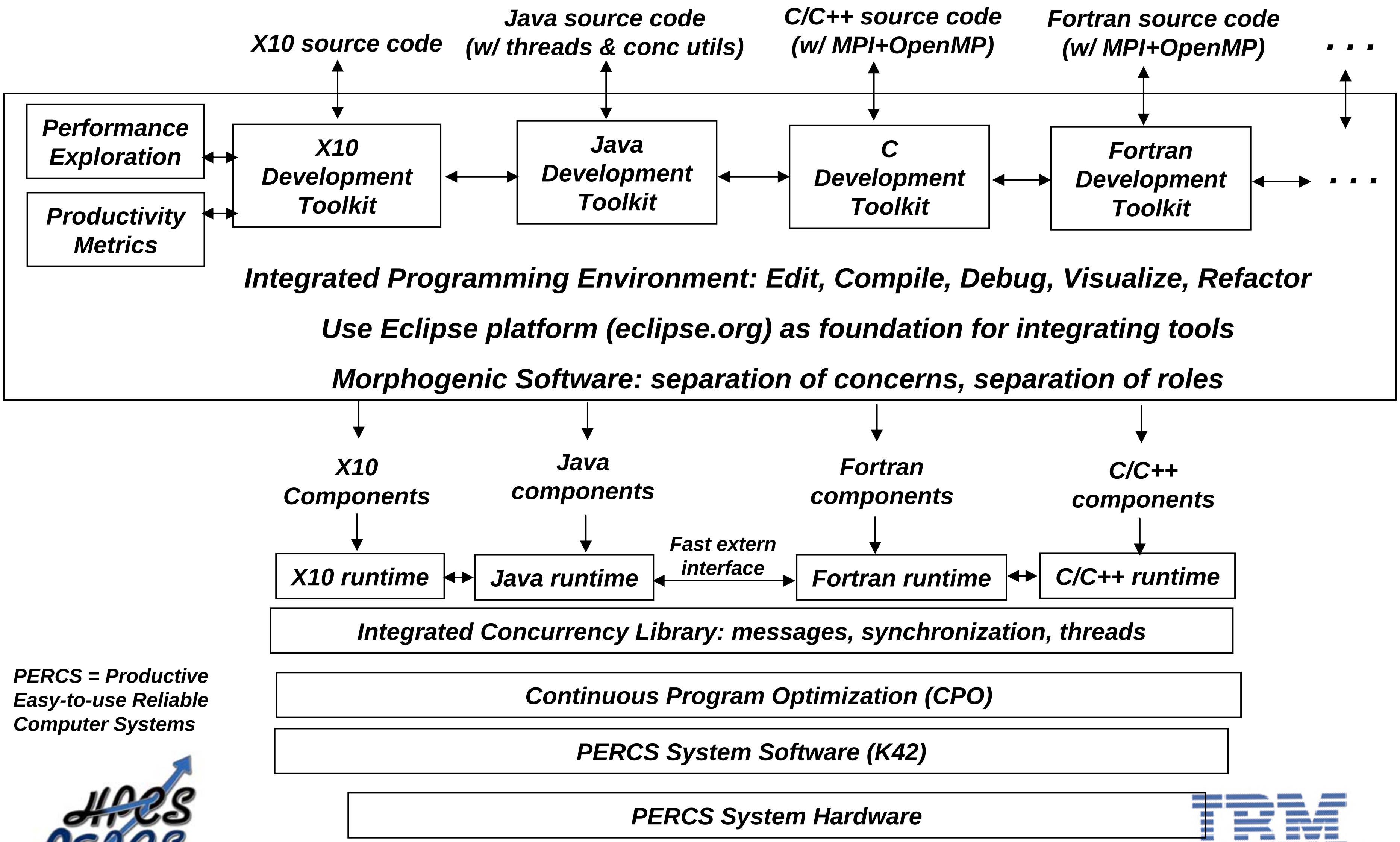
- Removes
  - Primitive arithmetic data types
  - Threads, lock-level synchronization
  - Single global heap
  - Arrays
  - JNI
- Adds
  - User-defined value types
  - Asynchronous activities, with atomic sections
  - Places specifying affinity between data and computation
  - True, distributed, multi-dimensional arrays
  - New efficient native/extern code invocation mechanisms

X10, in comparison with MPI+OpenMP …

- **Removes**
  - Processes
  - Programmer-managed global data structures
  - Message passing w/ programmer-managed marshalling
    - Includes reductions
  - Low-level message envelopes
    - <source, destination, tag, communicator>
  - Barriers
  - OpenMP threads
  - Locks, critical sections
  - Affinity directives
  - INDEPENDENT directive

- **Adds**
  - Places
  - Partitioned Global Address Space
  - Asynchronous activities w/ objects and futures
    - Includes reductions
  - Strongly-typed invocations and return values (futures)
  - Clocks
  - Asynchronous activities
  - Atomic sections
  - “at” clauses
  - foreach, ateach statements

X10 Programming and Runtime Environments



X10 source code

Java source code (w/ threads & conc utils)

C/C++ source code (w/ MPI+OpenMP)

Fortran source code (w/ MPI+OpenMP)

. . .

Performance Exploration

Productivity Metrics

X10 Development Toolkit

Java Development Toolkit

C Development Toolkit

Fortran Development Toolkit

. . .

Integrated Programming Environment: Edit, Compile, Debug, Visualize, Refactor

Use Eclipse platform (eclipse.org) as foundation for integrating tools

Morphogenic Software: separation of concerns, separation of roles

X10 Components

Java components

Fortran components

C/C++ components

X10 runtime

Java runtime

Fast extern interface

Fortran runtime

C/C++ runtime

Integrated Concurrency Library: messages, synchronization, threads

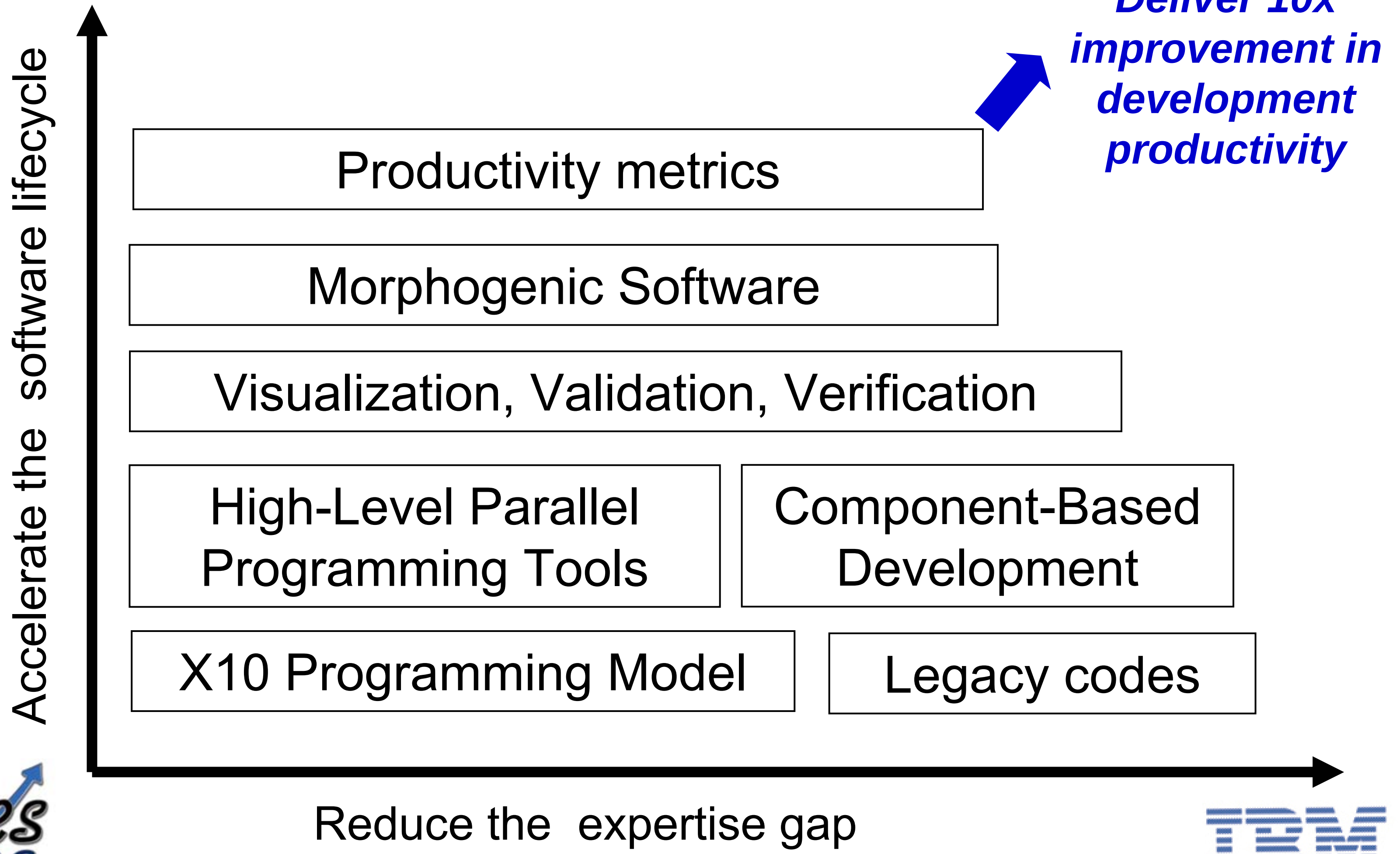
Continuous Program Optimization (CPO)

PERCS System Software (K42)

PERCS System Hardware

PERCS = Productive Easy-to-use Reliable Computer Systems

PERCS Programming Model and Tools:
Addressing Application Development Productivity Challenges



Accelerate the software lifecycle

Productivity metrics

Morphogenic Software

Visualization, Validation, Verification

High-Level Parallel Programming Tools

Component-Based Development

X10 Programming Model

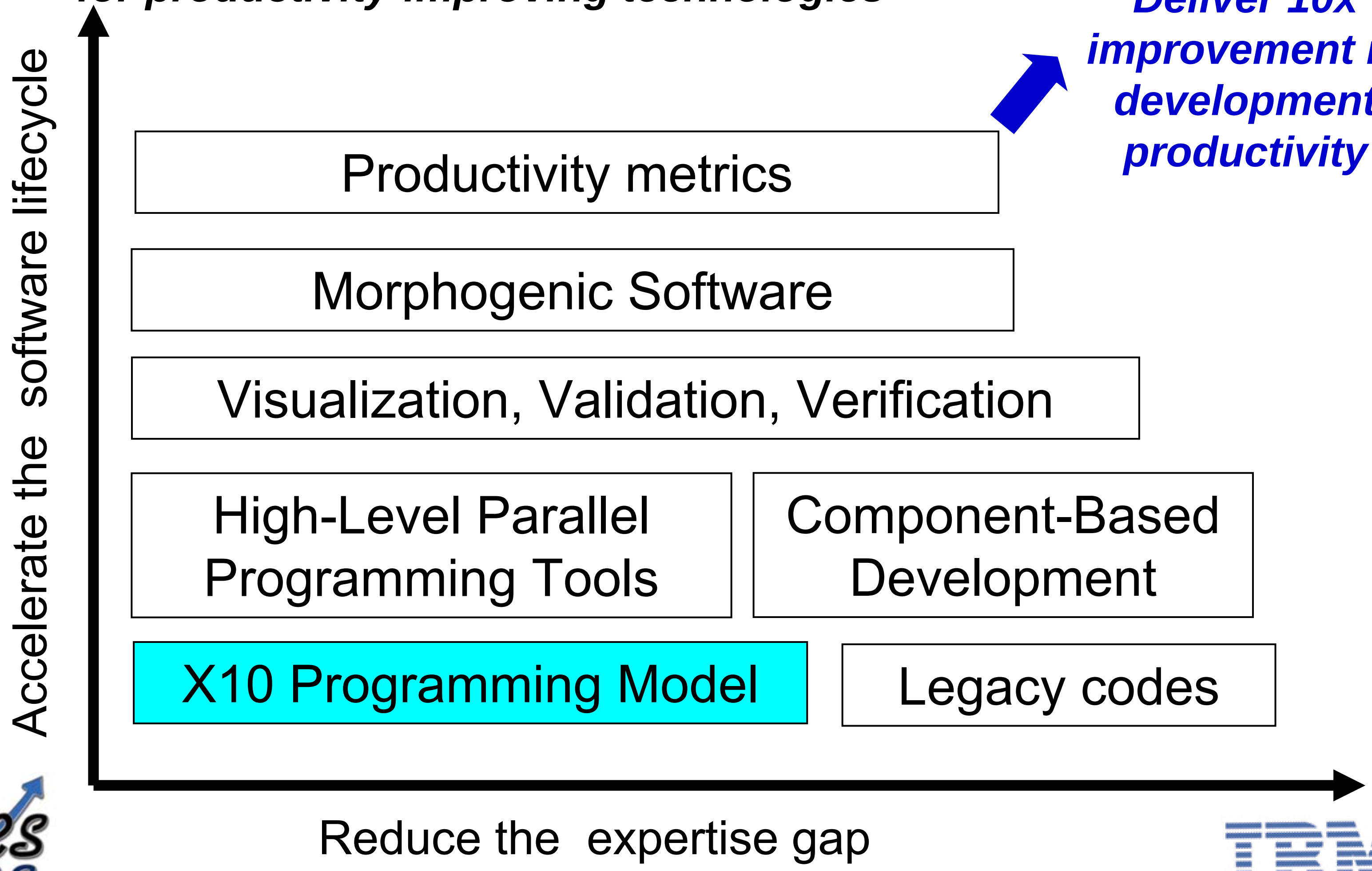
Legacy codes

Reduce the expertise gap

Grand challenge:
Deliver 10x improvement in development productivity

PERCS Programming Model and Tools: Addressing Application Development Productivity Challenges

New programming model provides foundation for productivity-improving technologies



Productivity metrics

Morphogenic Software

Visualization, Validation, Verification

High-Level Parallel Programming Tools

Component-Based Development

X10 Programming Model

Legacy codes

Accelerate the software lifecycle

Reduce the expertise gap

Grand challenge: Deliver 10x improvement in development productivity

Async activities: unified abstraction of threads and messages

- **Async statement (active message)**
  - `async(P){S}:` run S at place P
  - `async(D){S}:` run S at place containing datum `D`
  - S may contain local atomic operations or additional async activities for same/different places.
- *Example:*

```
public void put(K key, V value) {
   int hash = key.hashCode()% D.size;
  async (D[hash]) {
         for (_ b = buckets[hash]; b != null; b = b.next) {
            if (b.k.equals(key)) {
               b.v = value;
              return;
            }
         }
          buckets[hash] =
             new Bucket<K,V>(key, value, buckets[hash]);
    };
}
```

- **Async expression (future)**
  - `F = future(P){E}`, **or** `F = future(D){E}`: Return the value of expression E, evaluated in place P (or the place containing datum D)
  - `force F` **or** `!F :` suspend until value is known
- *Example:*

```
public ^V get(K key) {
     int hash = key.hashCode()% D.size;
     return future (D[hash]) {
        for (_ b = buckets[hash]; b != null; b = b.next) {
           if (b.k.equals(key)) {
              return b.v;
           }
        }
       return new V();
     }
  }
```

Distributed hash-table example

Clocks: abstraction of barriers

- **Operations:**

  `clock c = new clock();`

  `now(c){S}`

  - Require S to terminate before clock can progress.

  `continue c;`

  - Signals completion of work by activity in this clock phase.

  next $c_1,\ldots,c_n$ **;**

  - Suspend until clocks can advance. Implicitly continues all clocks. $c_1,\ldots,c_n$ names all clocks for activity.

  `drop c;`

  - No further operations on c..

- **Semantics**
  - Clock c can advance only when all activities registered with the clock have executed continue c..
- **Clocked final**
  - clocked(c) final int l = r;
  - Variable is “final” (immutable) until next phase

RandomAccess (GUPS) example

```
public void run(int a[] blocked, int seed[] cyclic,
            int value smallTable[]) {
    ateach (start : seed) clock(c) {
       int ran = start;
       for (int count : 1.. N_UPDATES/place.MAX_PLACES) {
           ran = Math.random(ran);
           int j = F(ran); // function F() can be in C/Fortran
           int k = smallTable[g(ran)];
           async (a[j]) atomic {a[j]^=k;}
       } // for
    } // ateach
   next c;
}
```

Regions and Distributions

- **Regions**
  - The domain of some array; a collection of array indices
  - region R = [0..99];
  - region R2 = [0..99,0..199];
- **Region operators**
  - region Intersect = R3 && R4;
  - region Union = R3 || R4;
  - Etc.
- **Distributions**
  - Map region elements to places
    - distribution D = cyclic(R);
  - Domain and range restriction:
    - distribution D2 = D | R;
    - distribution D3 = D | P;
- Regions/Distributions can be used like type and place parameters
  - <region R, distribution D> void m(...)

ArrayCopy example: example of high-level optimizations of async activities

Version 1 (orginal):

```
<value T, D, E>  public static void
    arrayCopy( T[D] a, T[E] b) {
        // Spawn an activity for each index to
        // fetch and copy the value
       ateach (i : D.region)
           a[i] = async b[i];
       next c; // Advance clock
    }
```

Version 2 (optimized):

```
<value T, D, E>  public static void
    arrayCopy( T[D] a, T[E] b) {
       // Spawn one activity per place
       ateach ( D.places )
           for ( j : D | here )
              a[i] = async b[i];
       next c; // Advance clock
    }
```

Version 3 (further optimized):

```
<value T, D, E>  public static void
    arrayCopy( T[D] a, T[E] b) {
    // Spawn one activity per D-place and one
    // future per place p to which E maps an
    // index in (D | here).
       ateach ( D.places ) {
          region LocalD = (D | here).region;
          ateach ( p : E[LocalD] ) {
              region RemoteE = (E | p).region;
              region Common =
                                   LocalD && RemoteE;
              a[Common] = async  b[Common];
          }
       }
      next c; // Advance clock
    }
```

Uniform treatment of Arrays & Loops and Collections & Iterators

- **Arrays**
  - Map region elements to values (therefore multidimensional)
  - Declared with a given distribution
  - int[D] array;
- **Loops**
  - ateach (D[R]) { ... }
  - ateach (array) { ... }
  - foreach (i : R) { ... }
  - foreach (i : D) { ... }
  - foreach (i : array) { ... }
  - sequential variants of foreach are available as for loops
- **Distributed Collections**
  - Map collection elements to places
  - Collection<D,E> identifies a collection with distribution D and element type E
- **Parallel iterators**
  - foreach (e : C) { … }
  - ateach (C) { … here … }
- **Sequential iterator**
  - for (e : C)

Reduction and Scan Operators

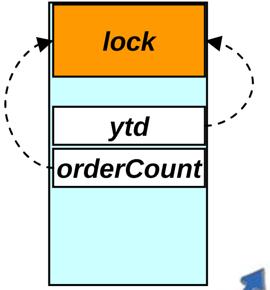
- Reduction operator over type T
  - Static method with signature: T(T,T)
  - Virtual method in class T with signature T(T)
  - Operator is expected to be associative and commutative
- Reduction operation: A >> foo() returns value of type T, where
  - A is an array over base type T
  - A>>foo() performs reductions over all elements of A to obtain a single result of type T
- Scan operation: A || foo() returns array, B, of base type T, where
  - B[i] = A[0..i]>>foo()

Example of Unconditional Atomic Sections SPECjbb2000: Java vs. X10 versions

Java version:

```
public class Stock extends Entity {…
private float  ytd;
private short orderCount; …
public synchronized void
  incrementYTD(short ol_quantity) { …
      ytd += ol_quantity; …}…
public synchronized void
  incrementOrderCount() { …
      ++orderCount; …} …
}
```

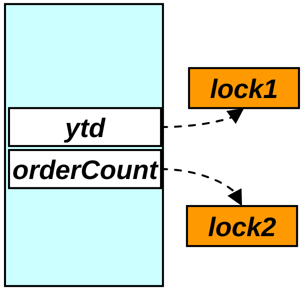
Layout of a “Stock” object



These two methods cannot be executed simultaneously because they use the same lock

X10 version (w/ atomic section):

```
public class Stock extends Entity {…
private float  ytd;
private short orderCount; …
public atomic void
  incrementYTD(short ol_quantity) { …
      ytd += ol_quantity; …}…
public atomic void
  incrementOrderCount() { …
      ++orderCount; …} …
}
```



With atomic sections, X10 implementation can choose to execute these two methods in parallel

Atomic Sections are deadlock-free!

Migrating Applications to X10

- OpenMP application
  - Can be initially implemented as single place w/ one activity per SPMD virtual processor
  - Partition into multiple places for improved performance
- Multithreaded applications
  - Can be initially implemented as single place w/ one activity per thread
  - Partition into multiple places for improved performance
- MPI
  - Partition into one place per processor
  - Replace message-passing operations by asynchronous operations

Relating optimizations for past programming paradigms to X10 optimizations

Programming paradigm	**Activities**	**Storage classes**	**Important optimizations**
Message-passing e.g., MPI	Single activity per place	Place local	Message aggregation, optimization of barriers & reductions
Data parallel e.g., HPF	Single global program	Partitioned global	SPMDization, synchronization & communication optimizations
PGAS e.g., Titanium, UPC	Single activity per place	Partitioned global, place local	Localization, SPMDization, synchronization & communication optimizations
DSM e.g., TreadMarks	Multiple	Partitioned global, activity local	Data layout optimizations, page locality optimizations
NUMA	Single activity per place	Partitioned global, activity local	Data distribution, synchronization & communication optimizations
Co-processor e.g., STI Cell	Single activity per place	Partitioned-global, place-local	Data communication, consistency, & synchronization optimizations
Futures / active messages	Multiple	Place-local, activity local	Message aggregation, synchronization optimization
Full X10	Multiple activities in multiple places	Partitioned-global, place-local, activity-local	All of the above

X10 Managed Runtime

Benefits of managed runtime systems and virtual machines are well understood …

- **Safety**
- **Productivity**
- **Portability**
- **Interoperability**
- **Isolation**
- **Virtualization**

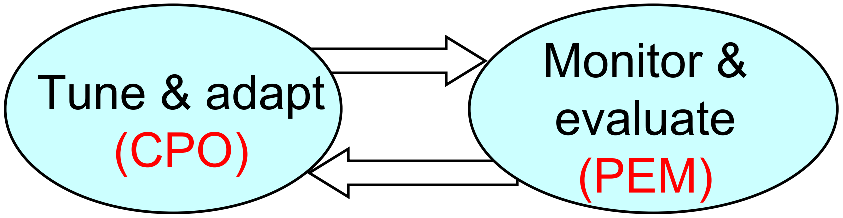
… but, are managed runtime systems appropriate for addressing performance challenges facing future large-scale parallel systems?

➔ Yes, because they enable *continuous program optimization*

Continuous Program Optimization (CPO) through Performance & Environment Monitoring (PEM)

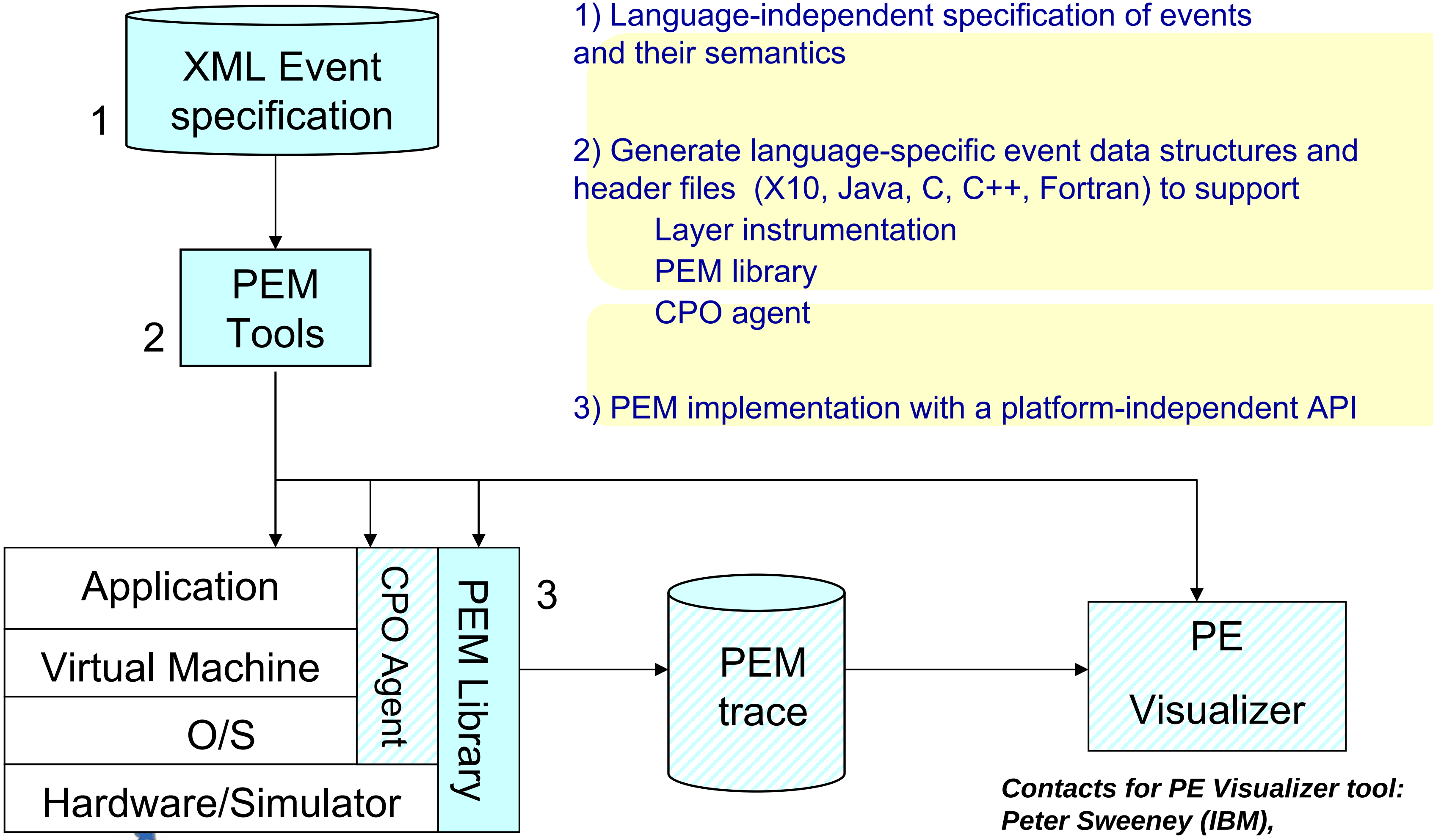
- Continuous Program Optimization (CPO) increases programmer productivity by automating the laborious and challenging performance tuning effort
- CPO aims at tuning application by optimally
  - **adapting the application to its behavior and environment**
  - **adapting environment resources to application behavior**
- CPO is made possible through continuous whole-system Performance and Environment Monitoring (PEM)

CPO/PEM contact:
Evelyn Duesterwald, IBM



Continuous optimization loop

PEM Infrastructure



1) Language-independent specification of events and their semantics

2) Generate language-specific event data structures and header files (X10, Java, C, C++, Fortran) to support

- Layer instrumentation
- PEM library
- CPO agent

3) PEM implementation with a platform-independent API

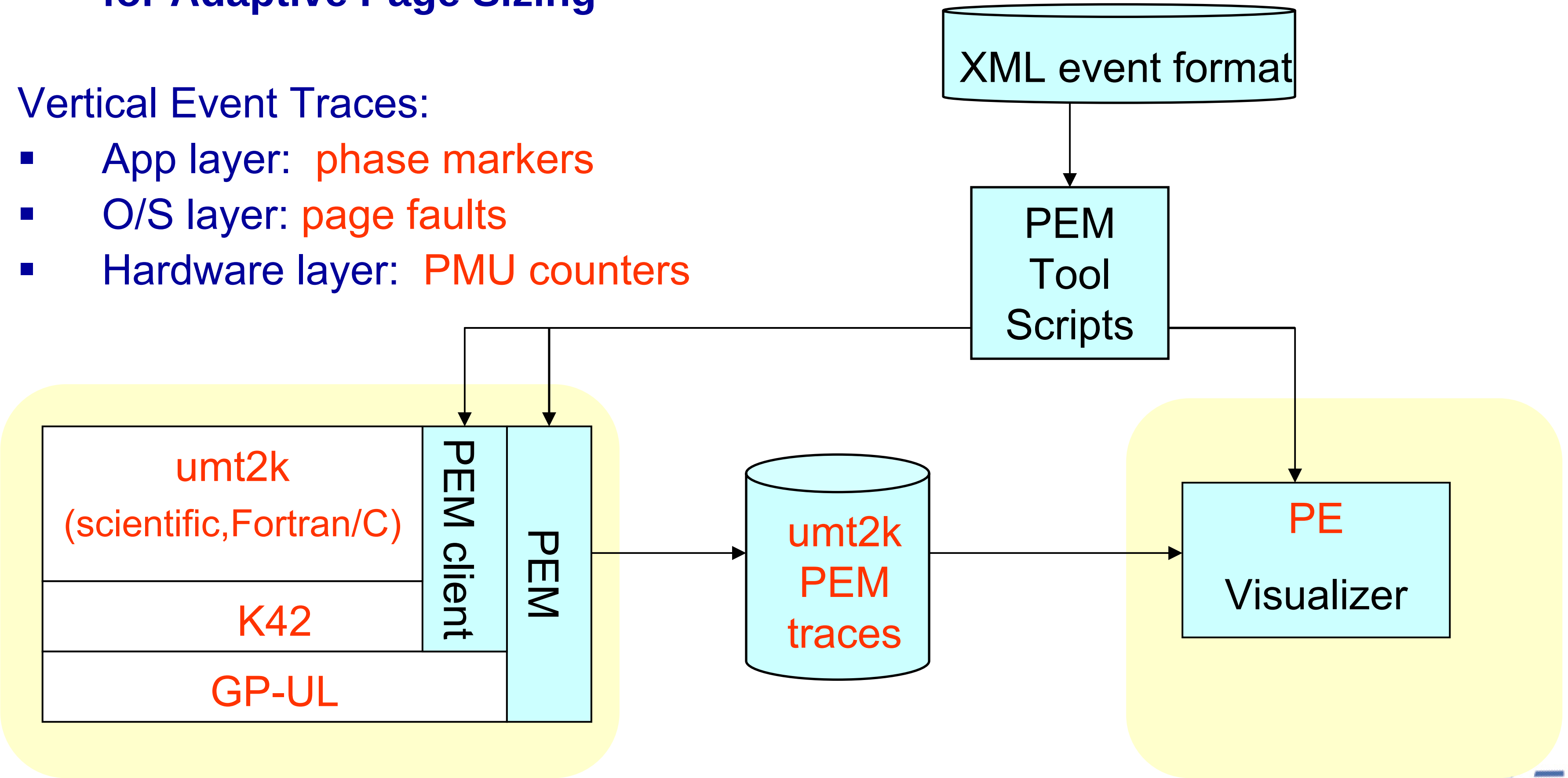
Contacts for PE Visualizer tool:
Peter Sweeney (IBM),
Matthias Hauswirth (U. Colorado)

PEM Scenario: Exploring the Performance Impact of Large Pages

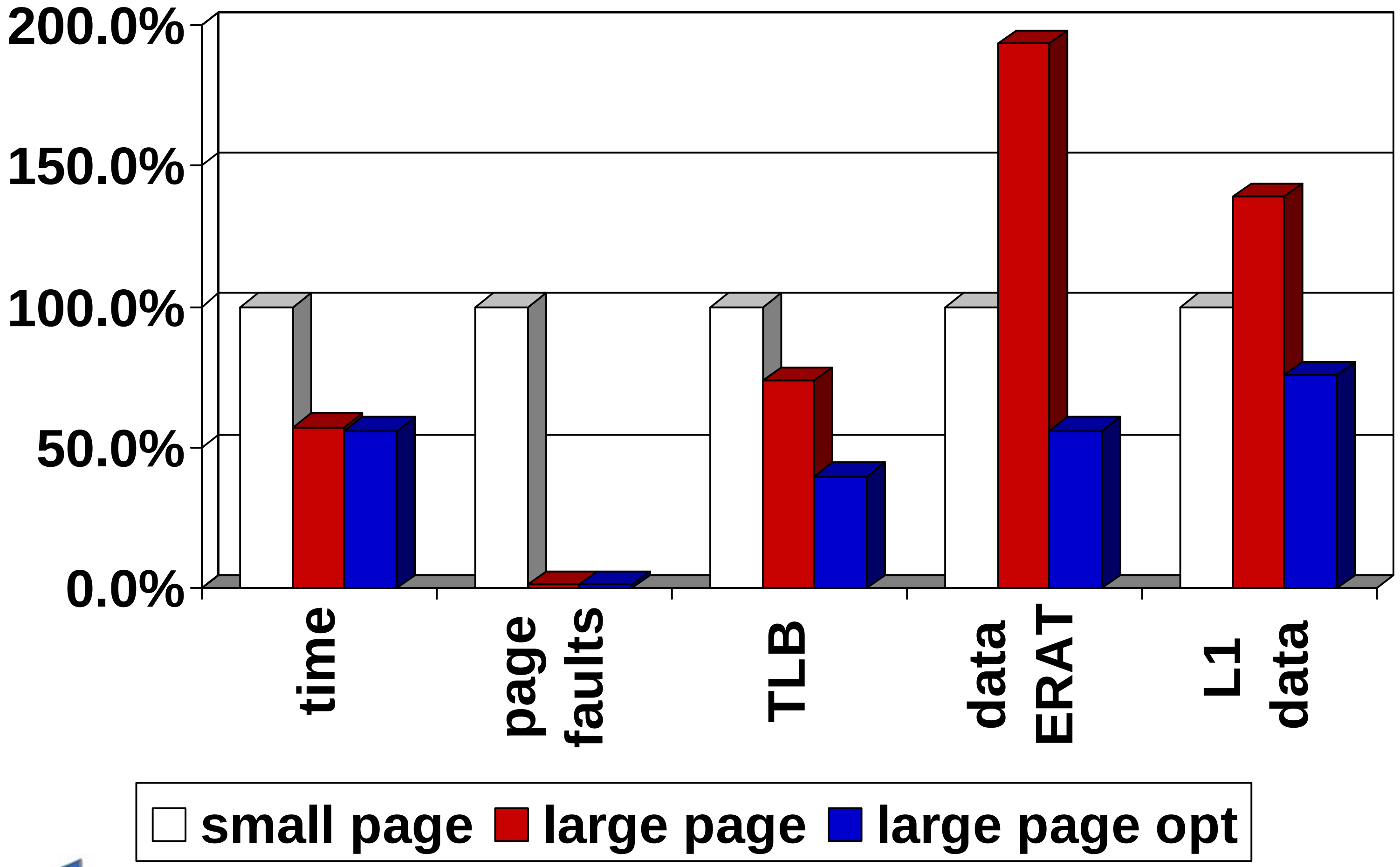
Preliminary Work for building a CPO Agent for Adaptive Page Sizing

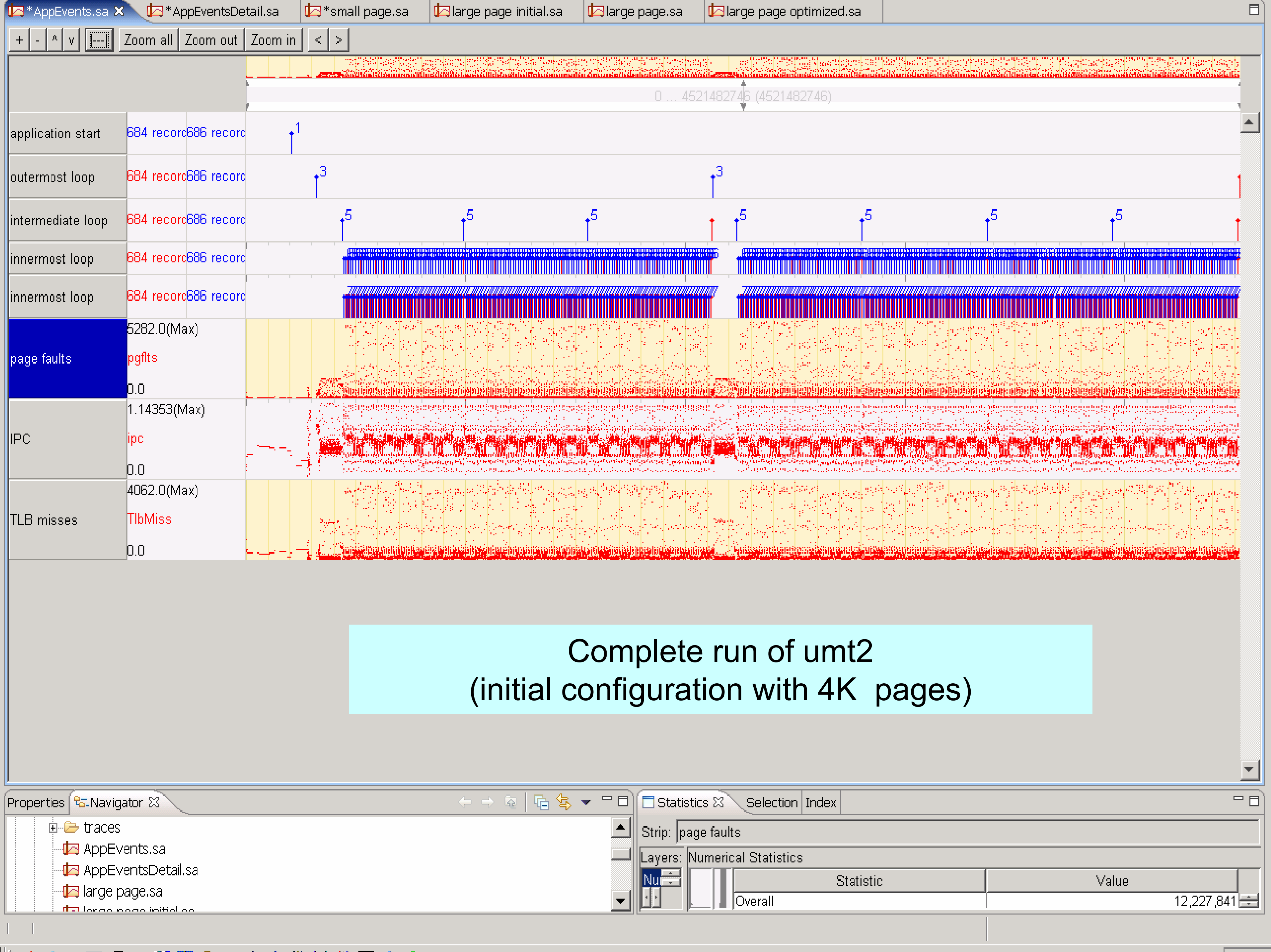
Vertical Event Traces:

- App layer: phase markers
- O/S layer: page faults
- Hardware layer: PMU counters



Summary of Performance Exploration



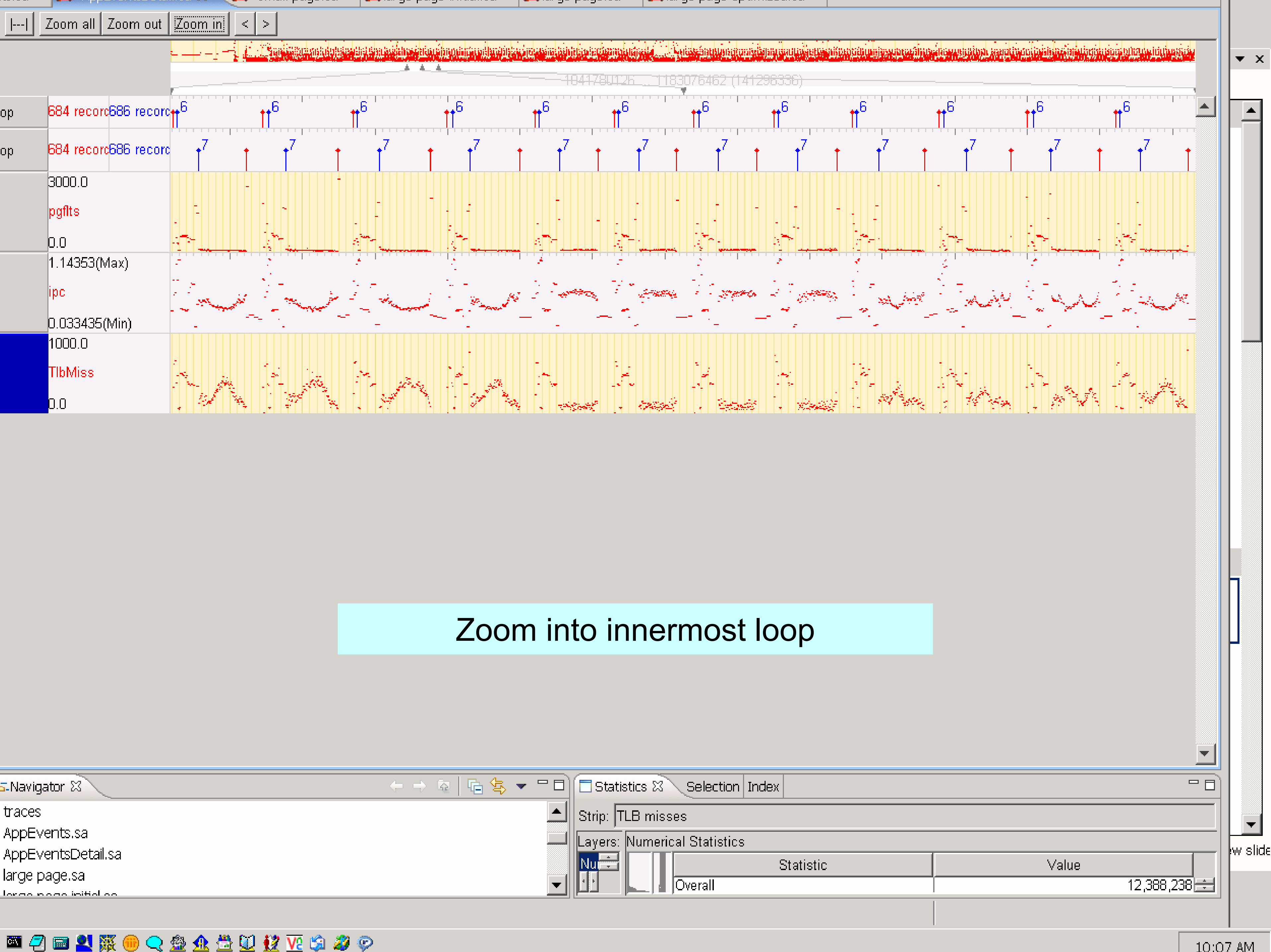


Strip: page faults

Layers: Numerical Statistics

Statistic	Value
Overall	12,227,841

Complete run of umt2
(initial configuration with 4K pages)

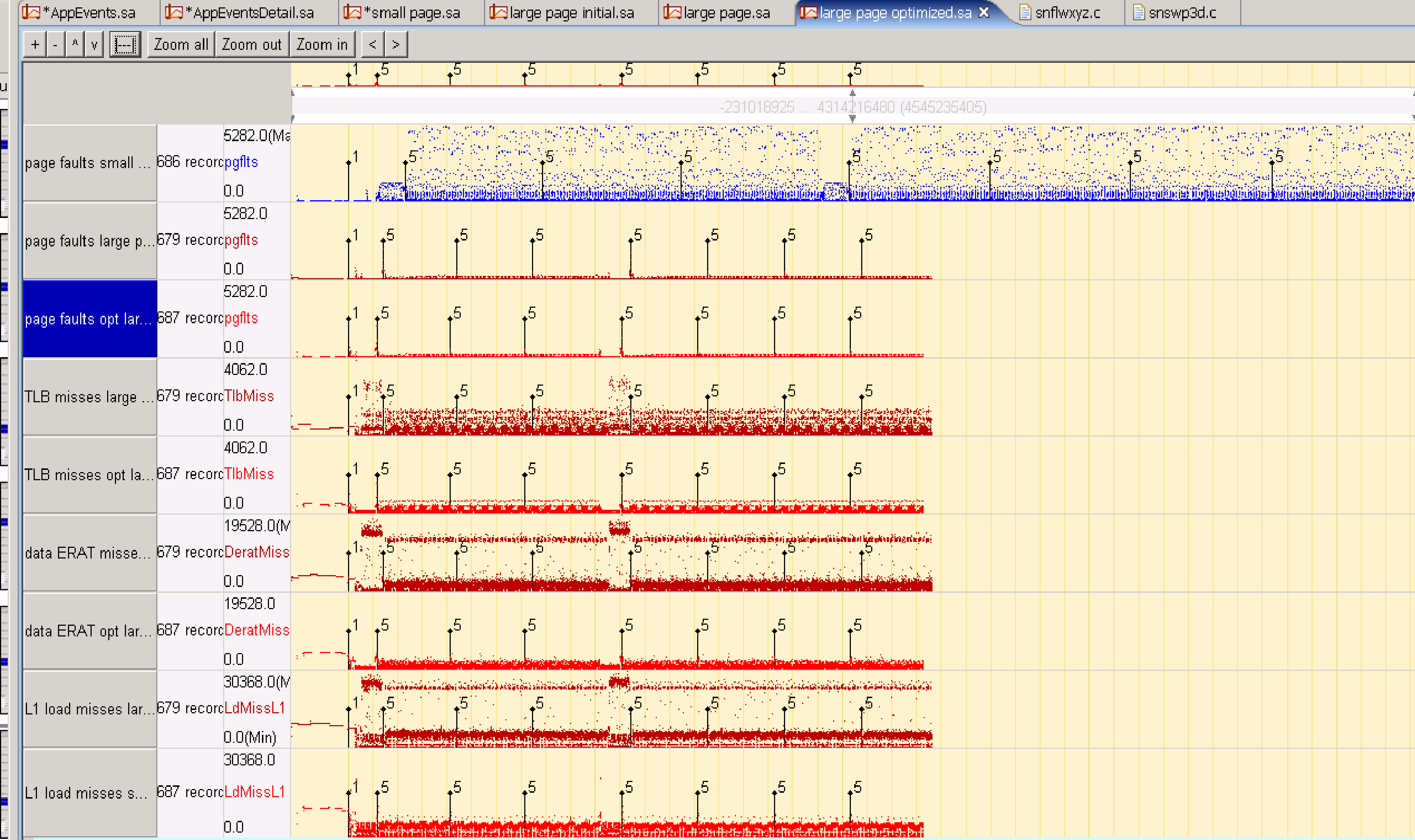


Strip: TLB misses

Layers: Numerical Statistics

Statistic	Value
Overall	12,388,238

Zoom into innermost loop



Blue – 4k pages

Brown – Initial Large Page Mapping: each structure aligned at large page boundary

Red - Optimized large page mapping: Offset each data structure to avoid conflicts

X10 Status and Plans

- Draft Language Design Report available internally w/ set of sample programs
- Implementation begun on X10 Prototype #1 for 1/2005
  - Functional reference implementation of language subset, not optimized for performance
  - Support for calls to single-threaded native code (C, Fortran)
- Productivity experiments planned for 7/2005
  - Use prototype #1 and related tools (PE, refactoring) to compare X10 w/ MPI, UPC
  - Revise language based on feedback from productivity experiments
- Prototype #2 planned for 12/2005
  - Includes design & prototype implementation of selected optimizations for parallelism, synchronization and locality in X10 programs
  - Revise language based on feedback from design evaluation

X10 Implementation Challenges

- **Type checking/inference** to enforce semantic guarantees
  - Clocked types
  - Place-aware types
- **Consistency management**
  - Lock assignment for atomic sections
  - Data-race detection
- **Activity aggregation**
  - Batch activities into a single thread.
- **Message aggregation**
  - Batch "small" messages.
- **Load-balancing**
  - Dynamic, adaptive migration of place from one processor to another.
- **Continuous optimization**
  - Efficient implementation of scan/reduce
- **Efficient invocation of components in foreign languages**
  - C, Fortran
- **Garbage collection across multiple places**

Conclusions and Future Work

- Future Large-scale Parallel Systems will be accompanied by severe productivity and performance challenges
  - ➔ Opportunity for Languages, Compilers, and Runtime technologies to have even greater impact on scalable systems than before
- Summarized X10 language approach in PERCS project, with a focus on next steps:
  - Use applications and productivity studies to refine design decisions in X10
  - Prototype solutions to address implementation challenges
- Future work (beyond 2005)
  - Community effort to build consensus on standardized "high productivity" languages for HPC systems in the 2010 timeframe
  - Explore integration of X10 ideas with other research language efforts under way in IBM
    - XJ, BPEL, …