

Compiler Analysis and Optimization Challenges for Atomic Sections

TRAMP 2007 Workshop Position Statement

Vivek Sarkar

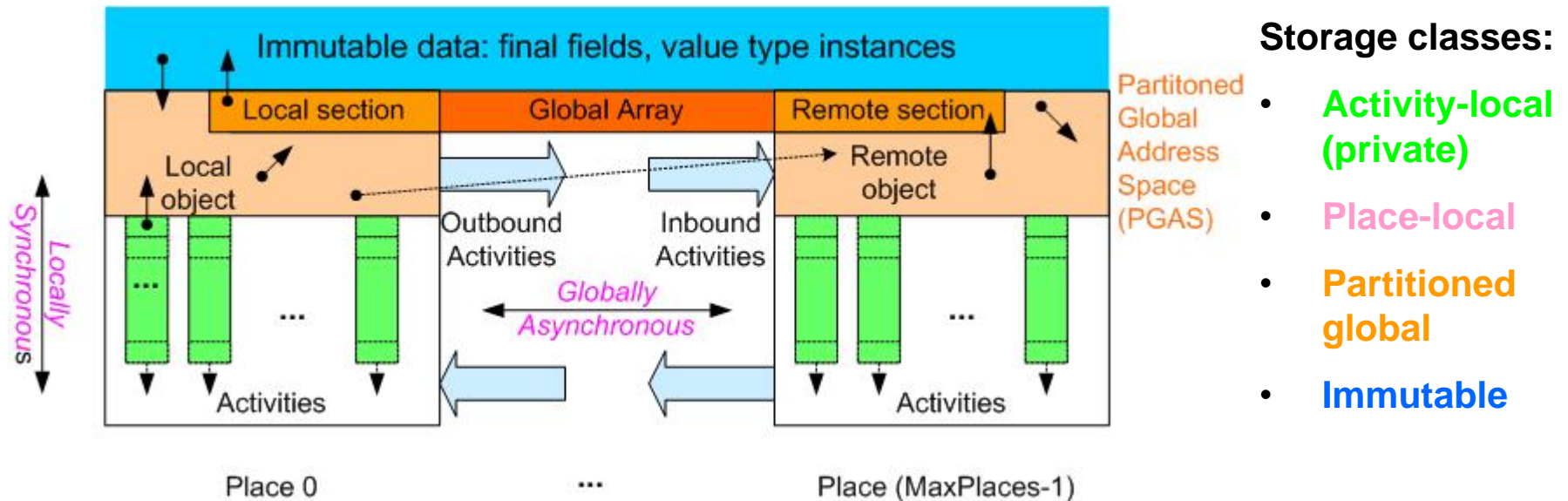
Senior Manager, Programming Technologies

IBM T.J. Watson Research Center

vsarkar@us.ibm.com



Context: X10 Programming Model



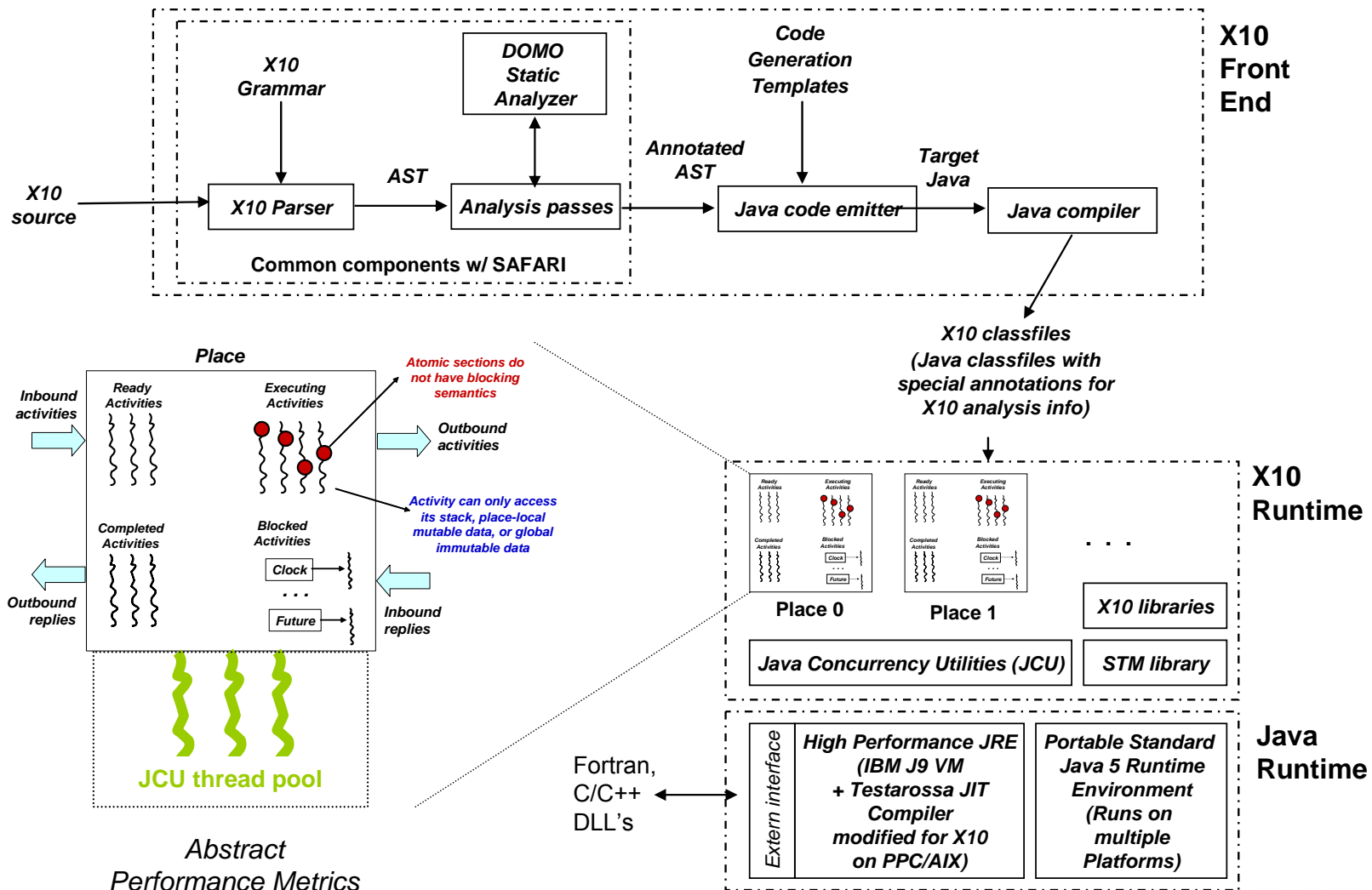
Storage classes:

- **Activity-local (private)**
- **Place-local**
- **Partitioned global**
- **Immutable**

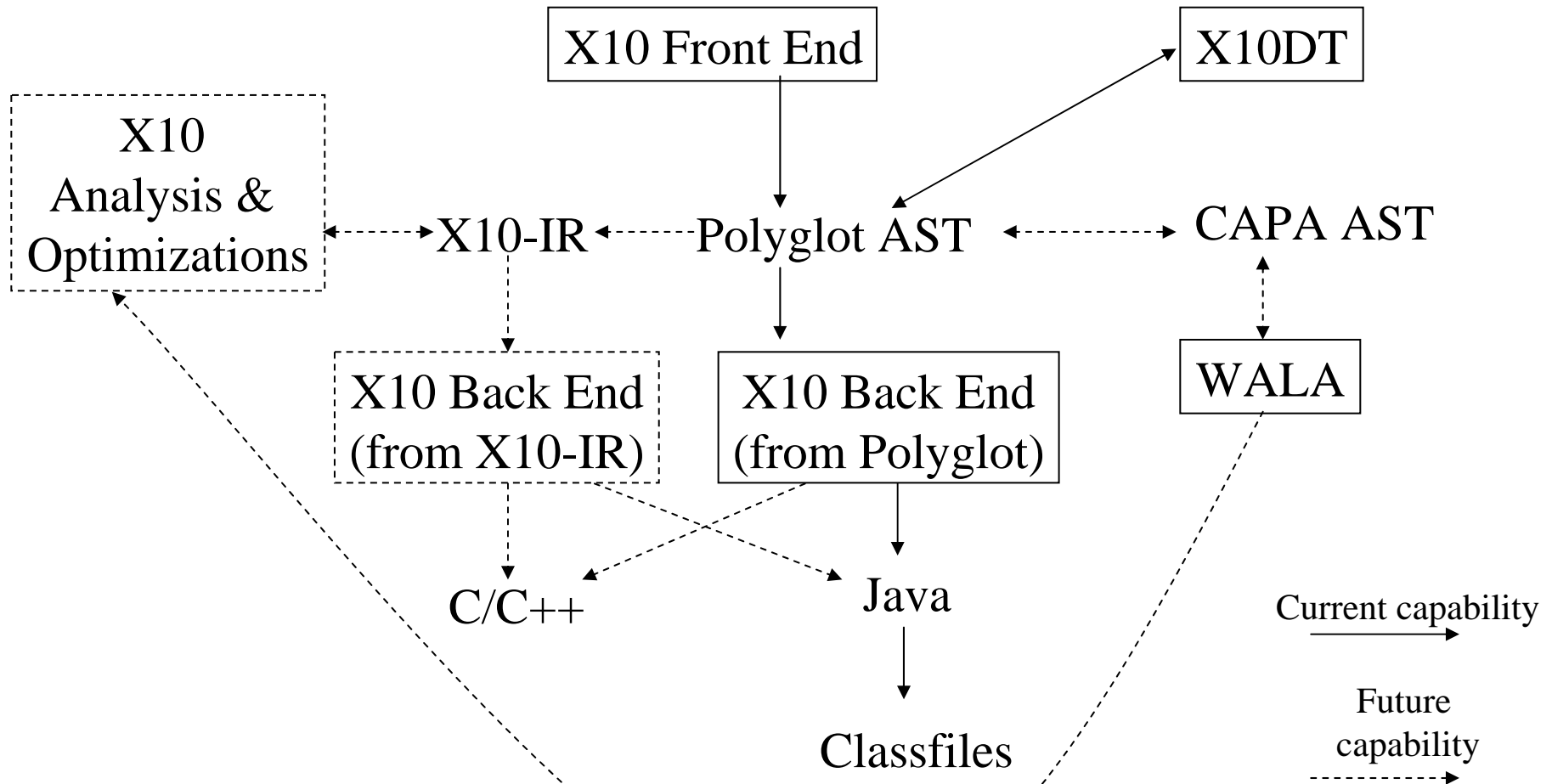
- Dynamic parallelism with a *Partitioned Global Address Space*
- *Places* encapsulate binding of activities and globally addressable data
- All concurrency is expressed as *asynchronous activities* – subsumes threads, structured parallelism, messaging, DMA transfers (beyond SPMD)
- **Atomic sections enforce mutual exclusion of co-located data**
 - **No place-remote accesses permitted in atomic section**
- *Immutable* data offers opportunity for single-assignment parallelism

Deadlock safety: any X10 program written with async, atomic, finish, foreach, ateach, and clocks can never deadlock

Open Source X10 SMP Implementation (x10.sf.net)



X10 compiler: future directions (Volunteers welcome!)



1) Program Analysis Challenges

- **How to model data flow & dependences for statements in atomic sections?**
 - e.g., analyses used in support of instruction scheduling and register allocation of heap data
- **How to partition data into different storage classes for atomicity?**
 - e.g., places, immutable, activity-local, single-owner, ...
- **How to identify code regions that can be implemented as atomic sections?**
 - automatic parallelization (reductions, commutativity), automatic generation of atomic sections from atomic sets
- **Note: program analysis for atomic sections can benefit both compilers and tools**

Example of May-Happen-In-Parallel Analysis of X10 [1]

```
/* Assume A has (BLOCK,BLOCK,*)
distribution */
```

```
for ( i = 1 ; i <= n ; i++ )
```

```
  finish
```

```
    for ( j = 1 ; j <= n ; j++ )
```

```
      for ( k = 1 ; k <= n ; k++ )
```

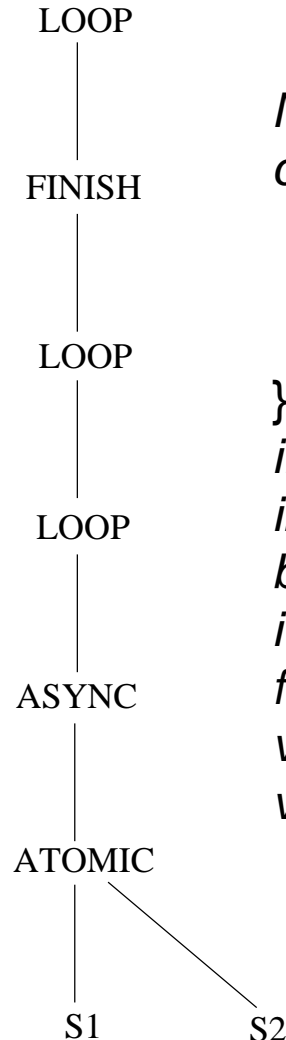
```
        async (A.distribution[i,j,k])
```

```
          atomic {
```

```
/* S1 */           temp = f(A[i,j,k]);
```

```
/* S2 */           A[i,j,k] = temp;
```

```
          }
```



MHP(S1 ,S2) = false with condition vector set, CS = {
<=, =, =>,
*<!=, *, *>,*
*<=, =, *>*
}

i.e., MHP(S1 ,S2) = false if instances of S1 and S2 belong to the same i-j-k iteration, or if they come from iterations with distinct values of i or with the same values of i and j

Dependence Analysis Example [4]

```

...
} // end of previous atomic

```

```

...
a = r.z
atomic {
  ... = p.x
  q.y = ...
  b =
}

```

```

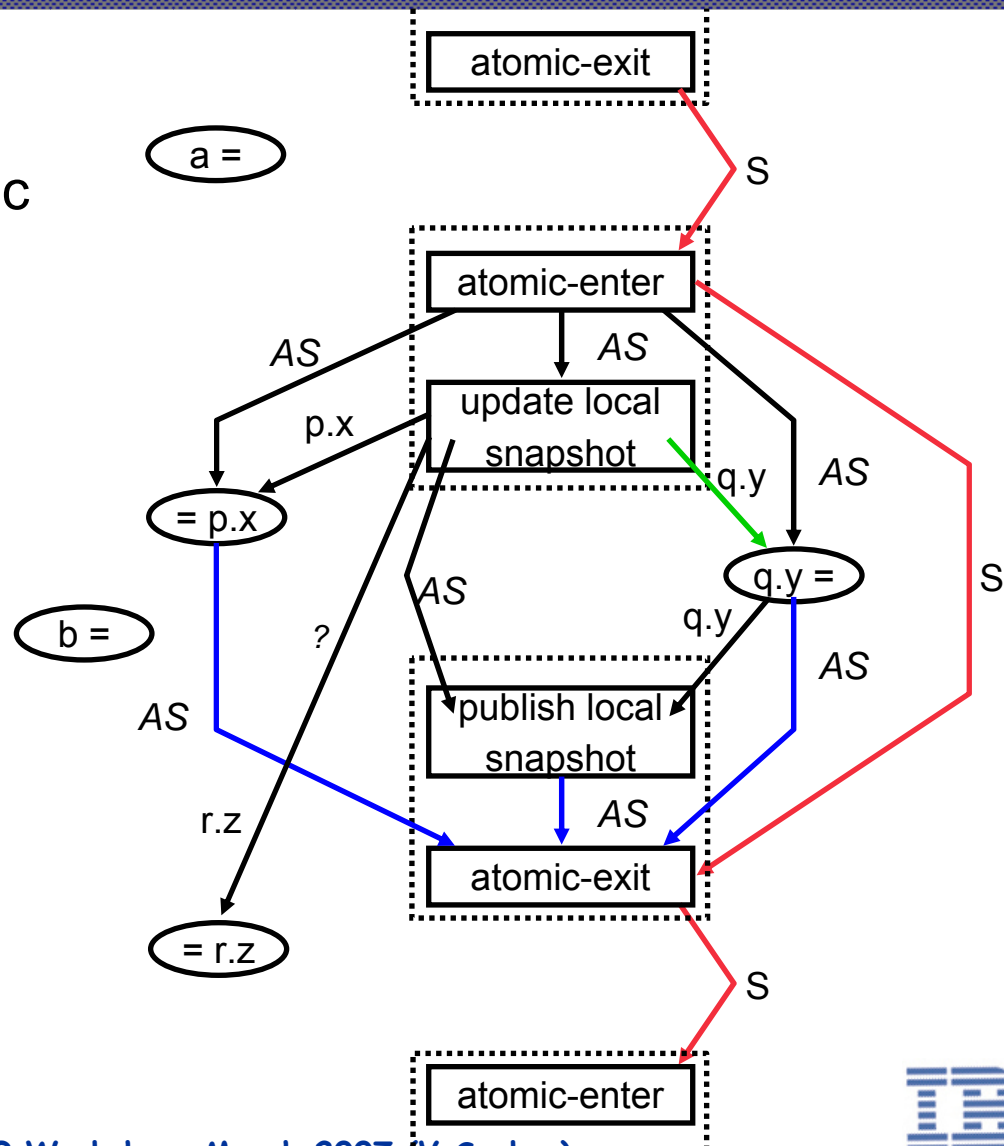
... = r.z

```

```

...
// start of next atomic
atomic {

```



2) Compiler Optimizations and Runtime Mechanisms

- **When can atomic section be implemented as a transaction?**
 - **Permissible operations, volume of data to be logged**
- **What transactional operations can be optimized by a compiler?**
 - **abort/retry, acquire/refresh, release/writeback**
- **When will automatic lock assignment be a better choice than a transactional memory implementation?**
- **How can thread scheduling policies help with optimized implementations of atomic sections?**
 - **Non-preemptive tasks, non-interruptible sections, eager vs. lazy execution of atomic sections**

Example of Compiler Optimization of Consistency Operations in Atomic Sections [18]

After insertion of consistency operations:

```
ATOMIC
  refresh(SCALE)
  IF (SCALE.LT.LSCALE) THEN
    refresh(SCALE)
    refresh(SSQ)
    SSQ= ((SCALE/LSCALE)**2)*SSQ+LSSQ
    writeback(SSQ)
    SCALE=LSCALE
    writeback(SCALE)
  ELSE
    refresh(SCALE)
    refresh(SSQ)
    SSQ=SSQ+ ((LSCALE/SCALE)**2)*LSSQ
    writeback(SSQ)
  ENDIF
  sync-writeback
END ATOMIC
```

After optimization of consistency operations:

```
ATOMIC
  refresh(SCALE)
  IF (SCALE.LT.LSCALE) THEN
    refresh(SSQ)
    SSQ= ((SCALE/LSCALE)**2)*SSQ+LSSQ
    writeback(SSQ)
    SCALE=LSCALE
    writeback(SCALE)
  ELSE
    refresh(SSQ)
    SSQ=SSQ+ ((LSCALE/SCALE)**2)*LSSQ
    writeback(SSQ)
  ENDIF
  sync-writeback
END ATOMIC
```

3) Atomic Sections and Global Collectives

- **Single Program Multiple Data (SPMD) programming model supports bulk-synchronous parallelism with coarse-grained collective operations**
- **Application trends (irregular) and hardware trends (multi-core) are creating pressure for finer-grained synchronization ... creating new opportunities for atomic sections and transactions in HPC code**
- **Key question:**
 - **When can a global reduction be split into fine-grained atomic reduction operations that preserve the original invariant?**

Summary

- **Compiler analysis and optimization of atomic sections and transactions depend critically on underlying semantics**
- **Three broad areas of research opportunity**
 - **Program Analysis**
 - **Compiler Optimizations and Runtime Mechanisms**
 - **Replacement for Global Collectives**