Compiler Analysis and Optimization of Habanero-Java Programs

Rajkishore Barik
rajbarik@rice.edu

Habanero Multicore Software Research Project

Rice University

Rice Habanero Multicore Software Project: Enabling Technologies for Extreme Scale

Parallel Applications

Portable execution model

1) Lightweight asynchronous tasks and data transfers
- *async, finish, asyncMemcpy*

2) Locality control for task and data distribution
- *hierarchical place tree*

3) Mutual exclusion
- *ownership-based isolation*

4) Collective, point-to-point, stream synchronization
- *phasers*

Habanero Programming Languages

Habanero Static Compiler & Parallel Intermediate Representation

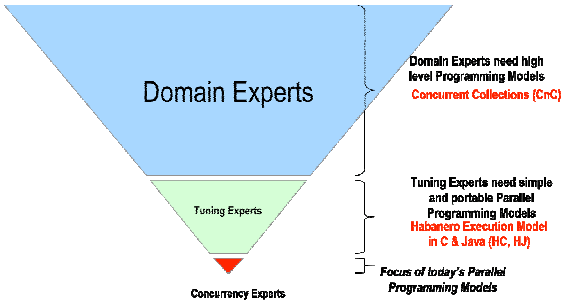
Habanero Runtime & Dynamic Compiler

Two-level programming model

Declarative Coordination Language for Domain Experts, CnC (Intel Concurrent Collections)

+

Task-Parallel Languages for Tuning Experts, Habanero-Java (from X10 v1.5) and Habanero-C



Extreme Scale Platforms

http://habanero.rice.edu

Code Optimization for Parallel Programs

- **Our current paradigm for code optimization was developed for sequential programs, and has served very well for five decades … but is now under siege because of parallelism**
- **Several anomalies can be observed when using sequential code optimization techniques on parallel programs**
  - **Control flow anomalies: branching due to parallel constructs**
    - **Arbitrary nesting of function calls and parallel constructs**
  - **Data flow anomalies: flow of values across parallel tasks**
    - **Shared data accesses may not be properly synchronized**
      - **Compiler does not know if input program is data-race free**
  - **Code motion anomalies: reordering of statements**
    - **Legality of the transformation depends on the underlying memory model supported by the programming language**

HJ Programming Model

- **Lightweight dynamic task creation & termination**
  - *async*: spawn an asynchronous activity
  - *finish*: parent activity waits for all children activities to complete
  - *future* async expressions and *force*
- **Mutual exclusion and isolation**
  - *isolated*: executed by an activity as if in a single step during which all other concurrent activities are suspended (extension of X10's atomic)
- **Collective and point-to-point synchronization**
  - *phasers* (extension of X10's clocks)
- **Locality control – task and data distributions**
  - Hierarchical *place* tree (extension of X10's places)
  - *Point, region,* and *distribution* of arrays
  - array views
- **Isolation Consistency Memory Model**

Habanero download website: *http://habanero.rice.edu/hj*

Async and Finish (from X10 v1.5)

Stmt ::= **async** *Stmt*

Stmt ::= **finish** *Stmt*

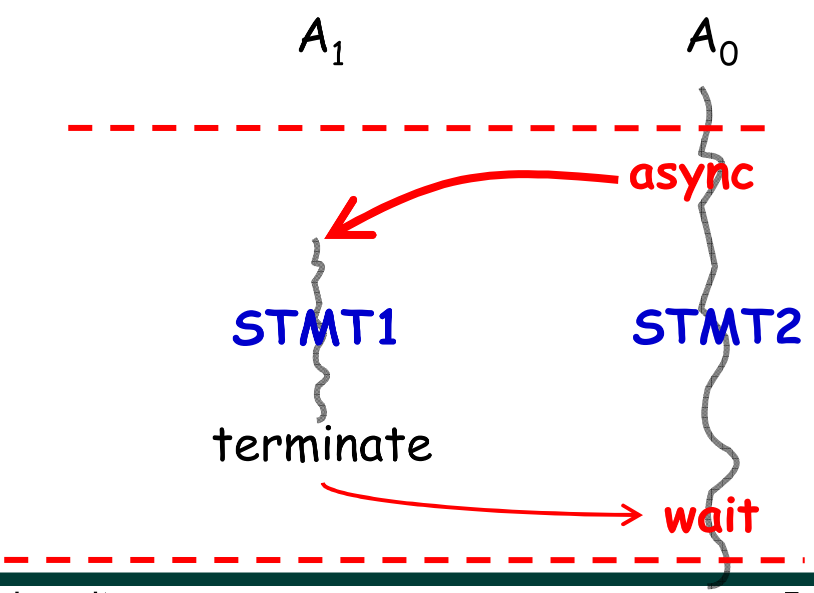
async S

- Creates a new child activity that executes statement S
- Returns immediately

finish S

- Execute S, but wait until all (transitively) spawned asyncs have terminated.
- Implicit finish between start and end of main program

```
//A0(Parent)
finish {    //Begin finish
  async {
    STMT1; //A1(Child)
  }
  STMT2;    //A0
}           //End finish
```



HJ isolated statement

isolated (<place-set>) <body>

isolated <body>

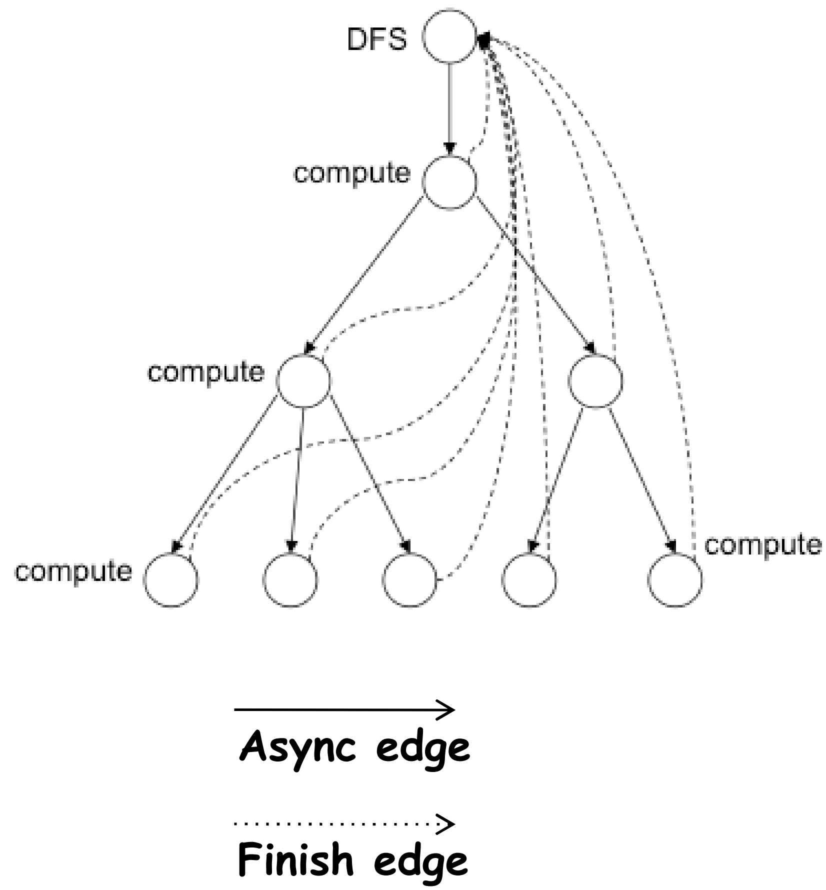
- Two tasks executing isolated statements with a non-empty place intersection must perform the isolated statement in mutual exclusion
- Tasks must only access data local to one of the places in <place-set>
  - Throw exception if a non-local access occurs
- Default: isolated = isolated(*), isolation across all places

Parallel Depth-First Search Spanning Tree

```
class V  {
  V [] neighbors;
  V parent;
  . . .
  boolean tryLabeling(V n) {
    isolated if (parent == null) parent = n;
    return parent == n;
  } // tryLabeling

  void compute() {
    for (int i=0; i<neighbors.length; i++) {
      V child = neighbors[i];
      if (child.tryLabeling(this))
          async child.compute(); //escaping async
     }
  } // compute

  void DFS() {
     parent = this; finish compute();
  } // DFS
} // class V
. . . root.DFS(); . . .
```



Scalar Replacement for Load Elimination

- **Scalar replacement for load elimination transformation replaces a heap memory load by a read of a scalar temporary**

  [Callahan et al '90, Cooper & Lu '97, Bodik et al '99, Wu & Lee '99, Fink et al '00, Cooper & Xu '02, Praun et al '03]

  - **Scalar replacement for register reuse leads to Load Elimination**
    - **Reuse using flow and input dependences**
  - **Needs reasoning about object references**

```
p   := new Type1()
q   := new Type1()
. . .
p.x  := p.x
q.x  := ....
...  := p.x
```

Original Code

```
p    := new Type1()
q    := new Type1()
. . .
T1    := p.x
p.x   := T1
q.x   := .....
......  := T1
```

Transformed code

Scalar Replacement Examples

Can the read in Line 4 reuse the value written in Line 2?

```
1: final A a = new A ()
2: a.f = …
3: async { … }
4: … = a.f
```

Case 1

```
1: final A a = new A ()
2: a.f = …
3: async { if(...) a.f = F(a.f) }
4: … = a.f
```

Case 2

```
1: final A a = new A ()
2: a.f = …
3: finish async { a.f = … }
4: … = a.f
```

Case 3

```
1: final A a = new A ()
2: a.f = …
3: async { isolated if (…) a.x++ }
4: … = a.f
```

Case 4

- **Legal for cases 1,2 and 4 in Isolation Consistency Memory model**

Summary of Scalar Replacement Algorithm

- **Eliminate GETFIELD operations across async, finish, and isolated constructs**
- **Compute Side-effects for every function call and parallel constructs (interprocedural analysis)**
- **Convert the program into Array-SSA form**
- **Perform scalar replacement using a data flow framework that propagates global value numbers**
- **Guarantee program semantics using Isolation Consistency Memory model that adheres to weak atomicity**

Interprocedural Load Elimination for Dynamic Optimization of Parallel Programs,
R. Barik, V. Sarkar, PACT 2009

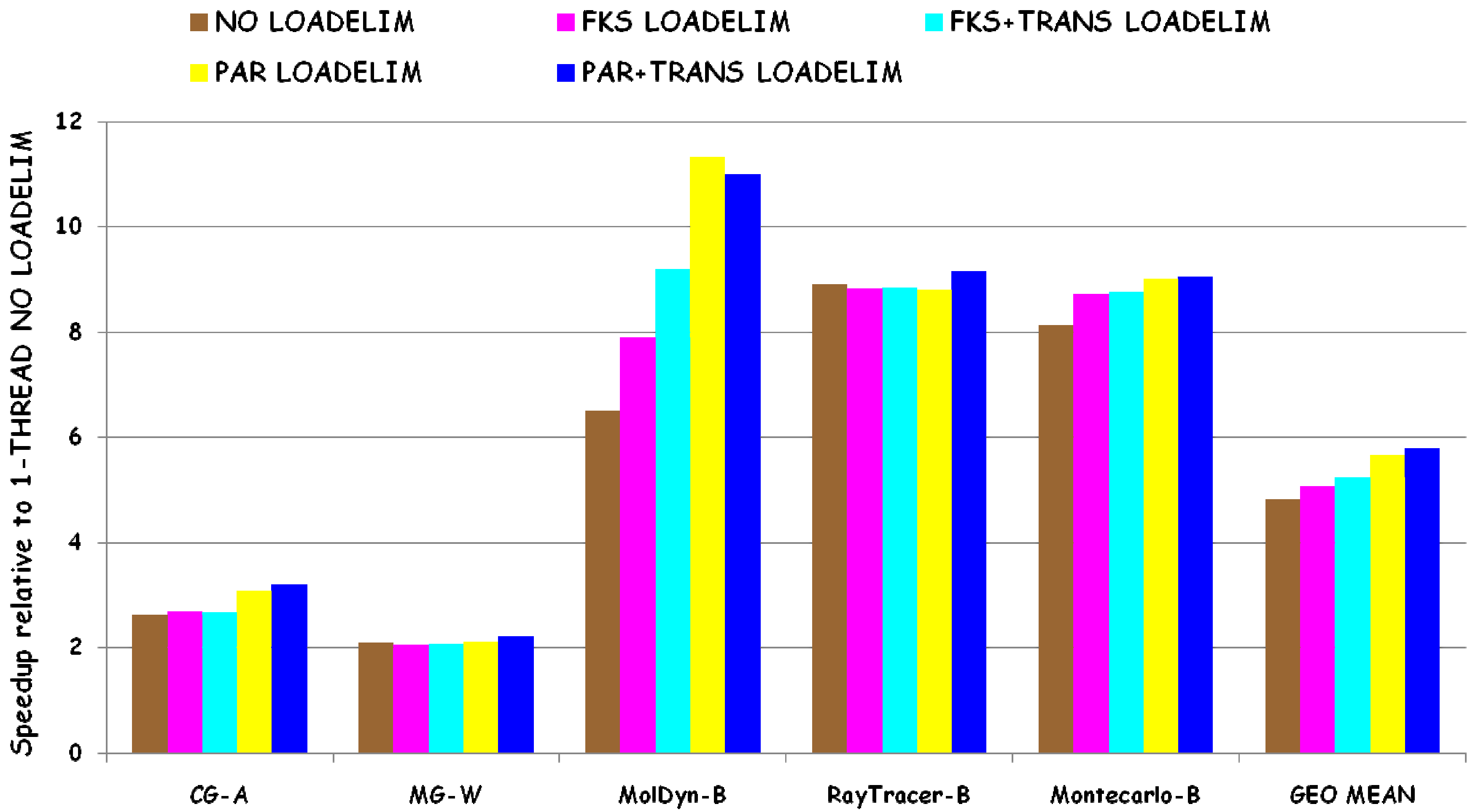
Experimental Setup

- **Hardware**
  - 16-core system that has four 2.4GHz quad-core Intel Xeon processors, 30GB of memory
- **Compiler and Runtime**
  - HJ front-end based on Polyglot
  - HJ middle-end based on Soot
  - Jikes RVM 3.0.0 with -X:aos:initial compiler=opt, -X:irc:O0, PLOS_FRAC=0.4f
  - HJ work-sharing runtime with NUMBER_OF_LOCAL_PLACES set to 1 and INIT_THREADS_PER_PLACE set to number of workers
- **Benchmark Set**
  - Java Grande Forum (Moldyn, Montecarlo, RayTracer)
  - NAS Parallel Benchmarks (CG, MG)

Experimental Setup (contd.)

- **Additional Transformations in Jikes RVM (TRANS):**
  - **Loop-invariant load motion**
    - **Convert while loops into zero-trip and a repeat-until loop**
  - **Live-range splitting**
    - **Split live-ranges around call and loop entry-exit regions**
- **Comparison of approaches (GETFIELD operations only):**
  - **Jikes RVM Load elimination (FKS)**
    - **Uses no side effect analysis for both function calls and parallel constructs**
  - **FKS with additional transformations (FKS+TRANS)**
  - **Parallelism-aware load elimination (PAR)**
  - **PAR with additional transformations (PAR+TRANS)**

Runtime Performance (16-Threads)

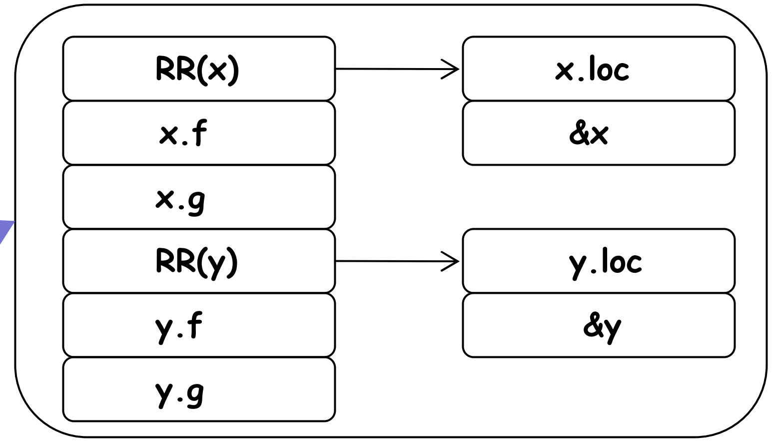


Speedup: up to 1.68x, and 1.22x on avg. compared to NO LOADELIM

Communication Optimization in X10

```
class C {
  val f,g;
  C (int m, int n) { f = m; g = n;}
}
```

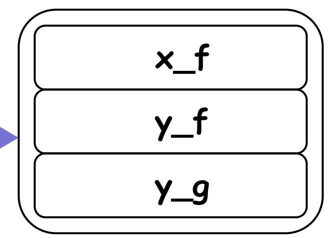
```
val x:C = new C(2,3);
val y:C = new C(2,3);
async (p) {
   = x.f;
   = y.f;
   = y.g
}
```



Communication Buffer

Transformed Program

```
val x:C = new C(2,3);
val y:C = new C(2,3);
val x_f = x.f, y_f=y.f, y_g=y.g;
async (p) {
   = x_f;
   = y_f;
   = y_g;
}
```



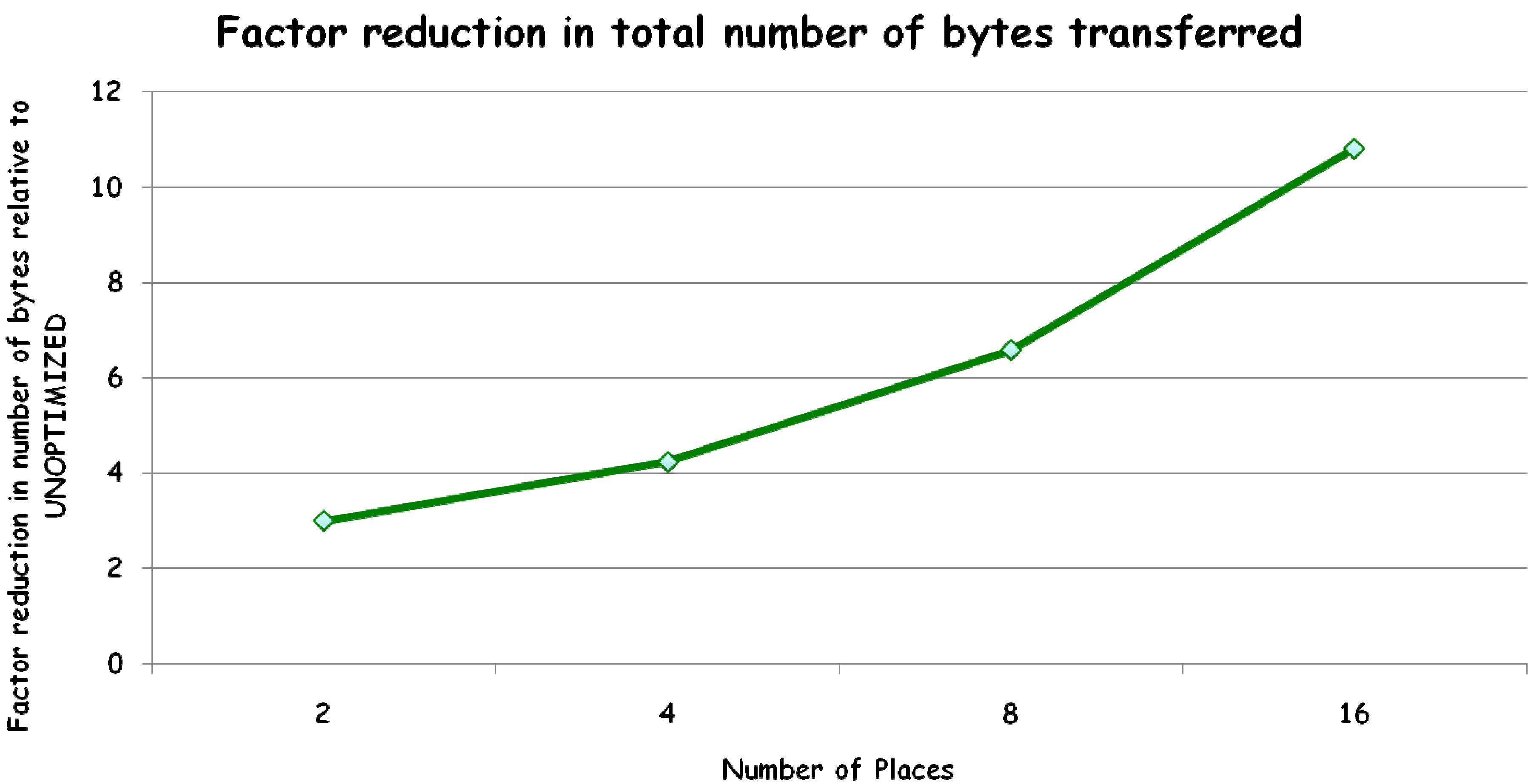
Communication Buffer

HPCC RandomAccess benchmark

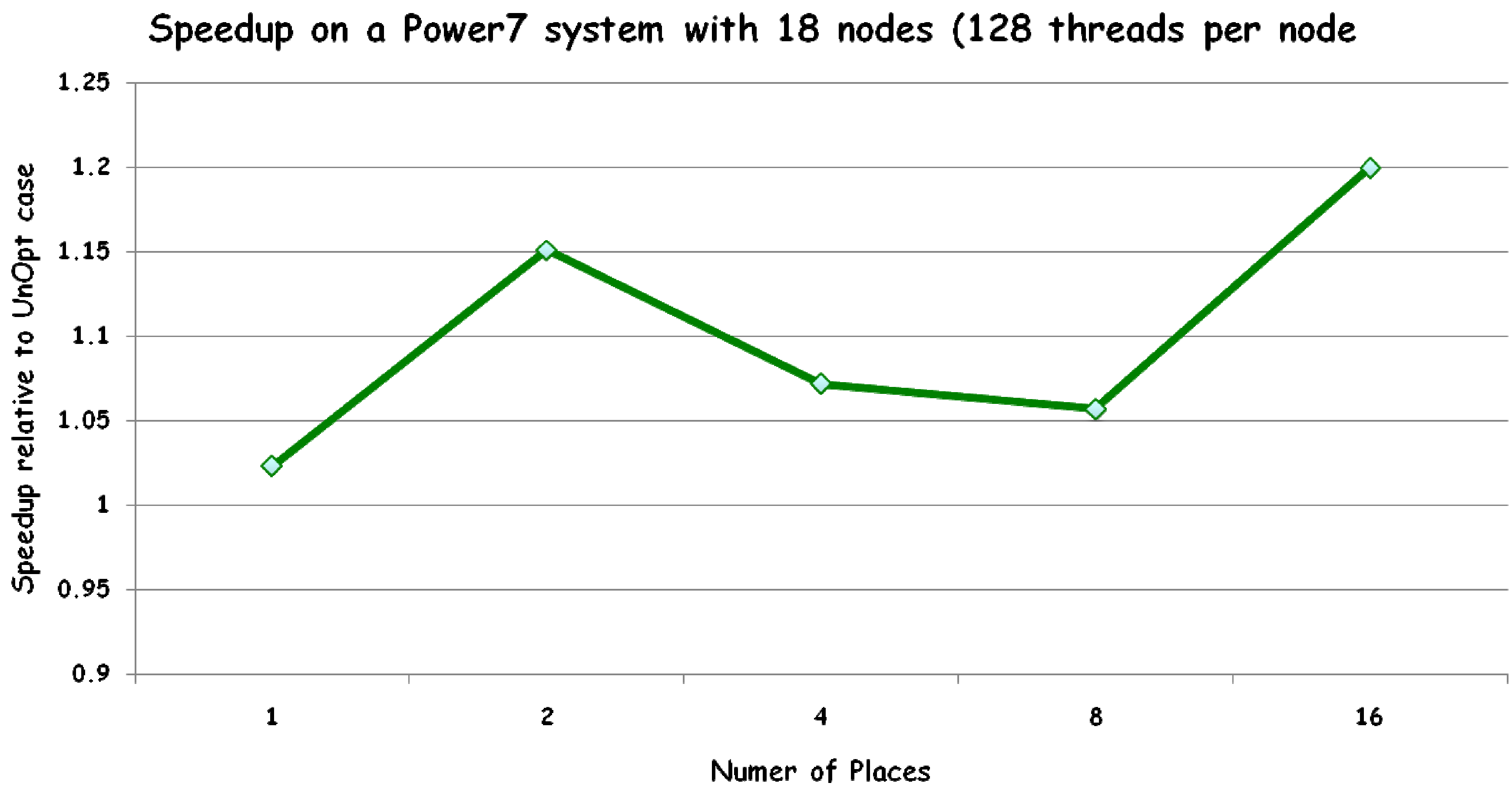
```
def randomAccessUpdate (NUM_UPDATES: long, logLocalTableSize: long,
    tables: ValRail[LocalTable]) {

  finish for (var p:int=0; p<Place.MAX_PLACES; p++) {
     val valp = p;
     async (Place.places(p)) { // async_0
       var ran:long = HPCC_starts(valp*(NUM_UPDATES/NUM_PLACES));
       for (var i:long=0; i<NUM_UPDATES/NUM_PLACES; i++) {
         val placeId = ((ran>>logLocalTableSize) & PLACE_ID_MASK) as int;
         val valran = ran;
         async (Place.places(placeId)) { // async_1
          tables(placeId).update(valran);
         }
         ran = (ran << 1) ^ (ran<0L ? POLY : 0L);
    } } } }
```

Preliminary results for RandomAccess



Preliminary results for RandomAccess



Conclusions and Future Work

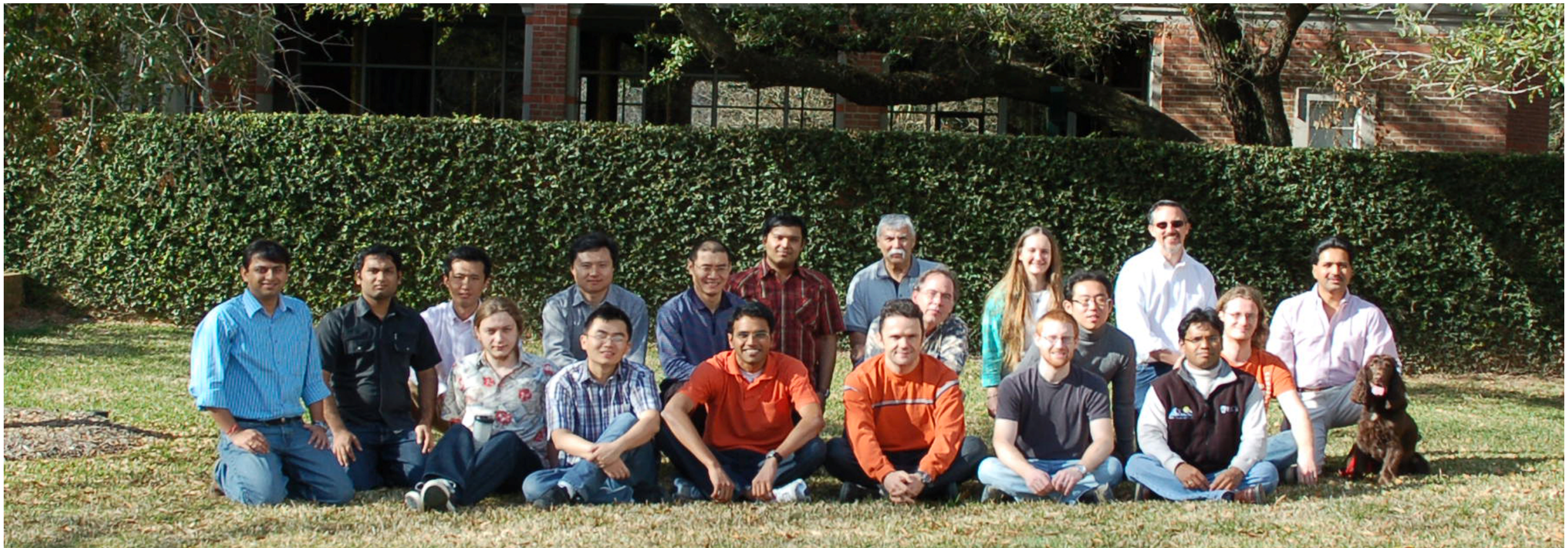
- **Habanero-Java (HJ) abstractions model important features of manycore systems including parallelism, synchronization, and locality**
- **Addressed compiler-level scalar replacement for load elimination in parallel programs with async, finish, and isolated constructs**
- **Extended scalar replacement optimization for communication optimization across async's in X10 programs on distributed-memory multiprocessors**

- **Future Work**
  - **Extend this work to additional parallel constructs like futures and phasers**
  - **Explore other compiler optimizations that are important for parallel program performance**

Acknowledgments

- Rice Habanero project team members
  - http://habanero.rice.edu
- IBM X10 project team members
- Contributors to open source projects
  - Jikes RVM, Polyglot, Soot, GCC
- Funding:
  - IBM Open Collaborative Faculty Award
  - This work was supported in part by the National Science Foundation under the HECURA program. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation.

Habanero Team

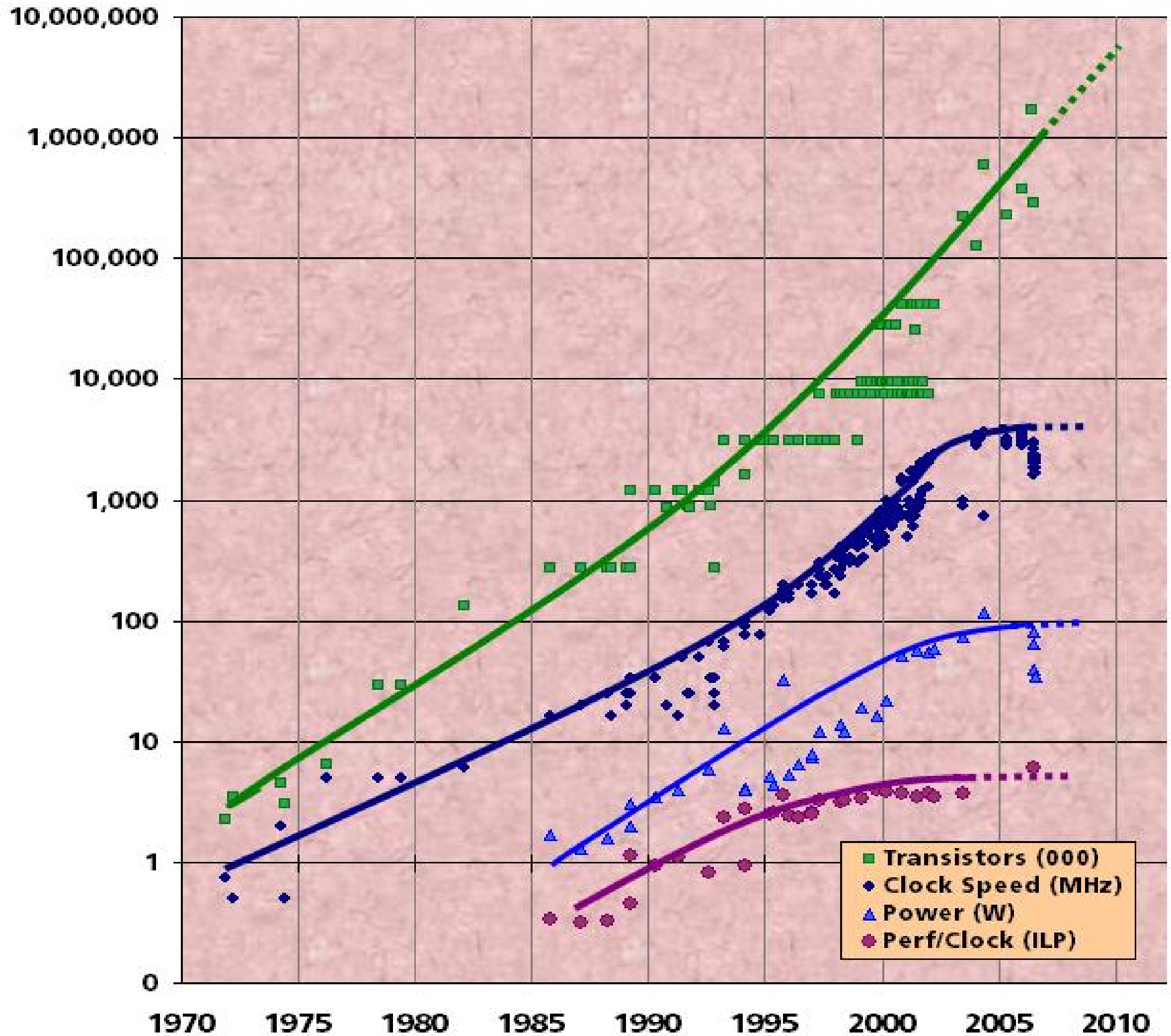


Thank You

Backup Slides

The Manycore Revolution: why Concurrency has become critical for Mainstream Computing

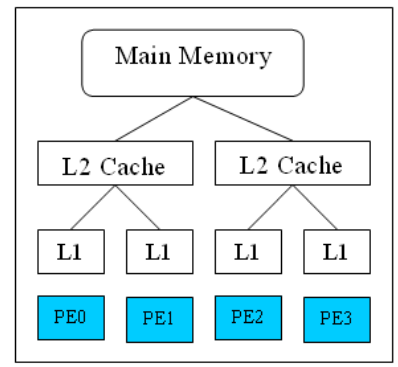
- **Chip density is continuing to increase ~2x every 18 months**
  - **Clock speed is not**
  - **Number of processor cores is doubling instead**
- **There is little or no hidden parallelism (ILP) to be found**
- **Manycore design with low power and area**
- ***Parallelism must be exposed to and managed by software explicitly***



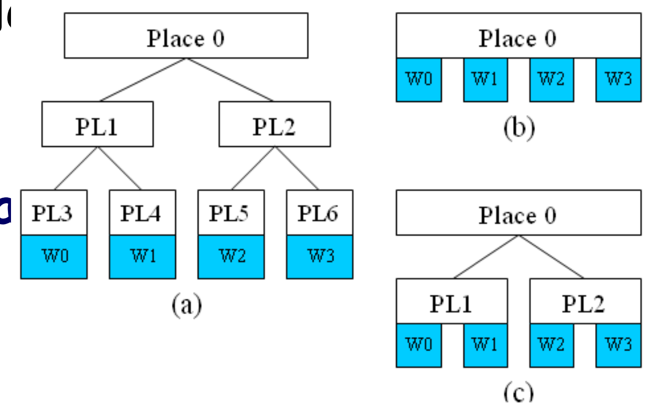
Source: Intel, Microsoft (Sutter) and Stanford (Olukotun, Hammond), Rice

Hierarchical Place Trees (HPT)

- Past approaches
  - Flat single-level partition e.g., HPF, PGAS
  - Hierarchical memory model with static parallelism e.g., Sequoia
- HPT approach
  - Hierarchical memory + Dynamic parallelis
- Place denotes memory hierarchy level
  - Cache, SDRAM, device memory, …
- Leaf places include worker threads
  - e.g., W0, W1, W2, W3
- Places can be used for CPUs and accel
- Multiple HPT configurations
  - For same hardware and programs
  - Trade-off between locality and load-bala



A Quad-core workstation



"Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement", Y.Yan et al, LCPC 2009

Data transfers in HPT

Three data transfer interfaces:

1. ***Implicit data transfer through data distribution***
   - Data can be distributed (e.g., block/cyclic) at each level of hierarchical place tree
   - e.g., use to model hierarchical shared memories
2. ***Explicit data transfer using synchronous copy-in / copy-out***
   - Syntax: async [<*pl*>] IN (…) OUT (…) INOUT (…) <*stmt*>
   - e.g., used to model memory-to-memory transfers for accelerators such as GPGPUs
3. ***Explicit data transfer using asynchronous memory copy***
   - Syntax: asyncMemcpy(dest, src);
   - e.g., use to model inter-processor DMA (direct memory access) with *finish* for termination

Overview of Phasers

- New synchronization construct designed to integrate
  - Asynchronous barriers
  - Asynchronous point-to-point synchronizations
  - Asynchronous collectives
  - Streaming computations
  - Dynamic parallelism (number of activities synchronized on phaser can vary dynamically)
- Support for "fuzzy barriers" and "single" statements
- Phase ordering property
- Deadlock freedom with "next" operations
- Amenable to efficient hierarchical implementation

- References
  - "Phasers: a Unified Deadlock-Free Construct for Collective and Point-to-point Synchronization", J.Shirako, D.Peixotto, V.Sarkar, W.Scherer, ICS 2008
  - "Phaser Accumulators: a New Reduction Construct for Dynamic Parallelism", J.Shirako, D.Peixotto, V.Sarkar, W.Scherer, IPDPS 2009
  - "Hierarchical Phasers for Scalable Synchronization and Reduction", J.Shirako, V.Sarkar, IPDPS 2010 (to appear)

Phaser Operations in Habanero Java

- **Phaser allocation**
  - **Phaser ph = new Phaser(mode)**
    - **Phaser ph is allocated with registration mode**
    - **Mode:**

SINGLE

SIG_WAIT(default)

SIGNAL   WAIT

Registration mode defines capability
There is a lattice ordering of capabilities

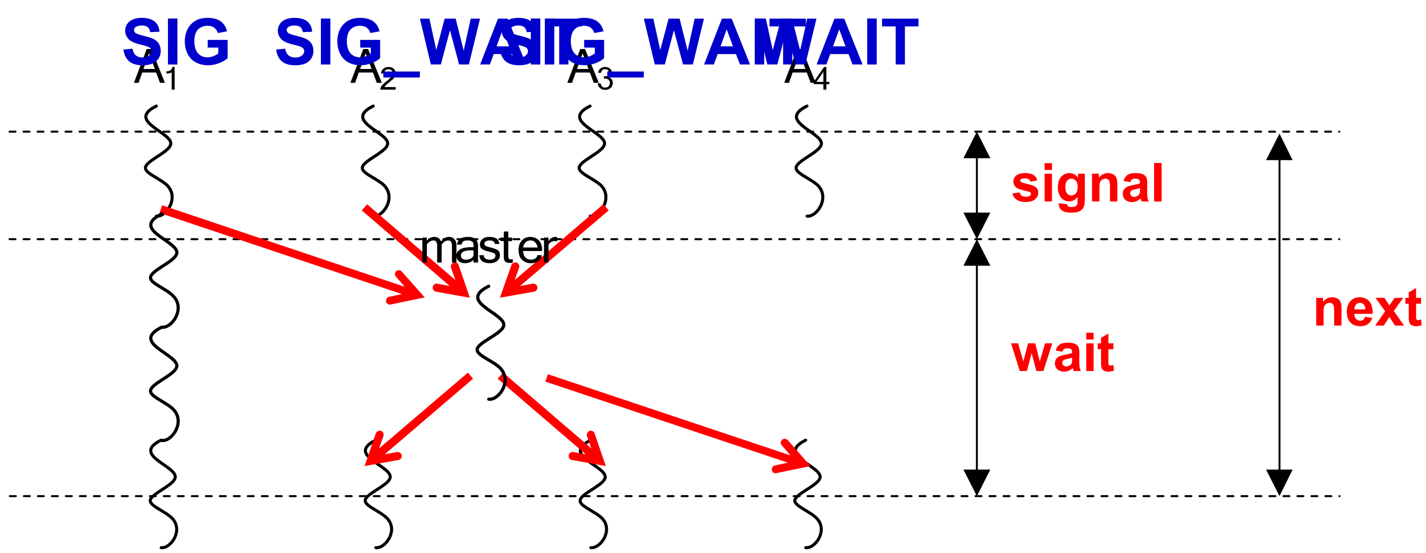
- **Activity registration**
  - **async phased (ph_1<$mode_1$>, ph_2<$mode_2$>, …) {STMT}**
    - **Spawned activity is registered with ph_1 in $mode_1$, ph_2 in $mode_2$, …**
    - **child activity's capabilities must be subset of parent's**
- **Synchronization**
  - **next:**
    - **Advance each phaser that activity is registered on to its next phase**
    - **Semantics depends on registration mode**

next / signal / wait operations

next = { Notify "I reached next" = signal (or ph.signal())
Wait for others to notify = wait

Semantics of next depends on registration mode

- SIG_WAIT: next = signal + wait
- SIG: next = signal (Don't wait for any activity)
- WAIT: next = wait (Don't disturb any activity)



A master activity receives all signals and broadcasts a barrier completion

HJ's *Async* and *Finish* Statements for Task Creation and Termination

async S

- Creates a new child task that executes statement S
- Parent immediately moves on to statement following the async
- If S refers to a local variable from an enclosing statement, that variable must be declared as **final**
- Child task cannot be aborted or cancelled
- Analogous to pthread_create()

finish S

- Execute S, but wait until *all* (transitively) spawned asyncs in S's scope have terminated.
- Implicit finish between start and end of main program
- Analogous to pthread_join(), but applied to all descendant tasks

Rooted exception model
Trap all exceptions thrown by spawned activities
Throw an (aggregate) exception if any spawned async terminates abruptly

Related Work

- Scalar Replacement in the context of array references [Callahan et al '90]
  - Used to improve register reuse
- Redundant memory load operations using Global Value Numbering approach [Bodik et al '99]
- Unified framework for analyzing memory loads of arrays and object field references [Fink et al '00]
  - Array-SSA form and global value numbering
  - Conservative assumptions for function calls, parallel constructs
- Load elimination in the presence of Java's exception and concurrency using PRE [Praun et al '03]
  - Conflict Analysis that guarantees SC model
- Improve Fink et al. work by encoding side-effect information in class files [Le et al '05]
  - Uses results of points-to analysis from SOOT infrastructure

Scalar Replacement for Load Elimination Example

```
void main() {
  p.x = …
  s.w = …
  finish { //f
    async { //async_1
      if (…) p.x = …
      isolated { q.y = …; … = q.y }
      … = p.x
    }
    … = p.x
    foo()
  }
  … = p.x
  … = s.w
}
```

```
void foo() {
  async bar() //async_2
  isolated { q.y = … }
  … = s.w
}
```

```
void bar() {
  r.z = …
  .. = r.z
}
```

Can be replaced by a scalar

Can not be replaced by a scalar

Side-effects of method calls and parallel constructs

- **Async and normal method level side-effect** [Banning '79]
  - **IMOD/IREF – immediate modified/referenced side-effects of individual statements**
  - **GMOD/GREF – generalized modified/referenced side-effects of method calls**
    - **Propagate side-effects over call graph nodes**
- **Isolated level side-effect**
  - **AMOD/AREF – modified/referenced side-effects for isolated blocks**
    - **Global side-effects or refined side-effects based on May-Happen-in-Parallel analysis**
- **Async-escaping level side-effect**
  - **EMOD/EREF – escaping modified/referenced side-effects**
- **Finish scope level side-effect**
  - **FMOD/FREF - modified/referenced side-effects for finish scope**

Isolation Consistency Memory Model

- **Isolation Consistency Memory Model**
  - Builds on the operational semantics of *Location Consistency* (LC) Memory Model [Gao & Sarkar '00]
    - State of a shared location is defined using a partially ordered multi-set (pomset) of write and synchronization operations
    - A read operation sees a value that is
      - written by a most recent predecessor write
      - a write operation that is unrelated
  - Weaker than many existing memory models including sequential consistency
    - Favors compiler optimization (like code motion) by preferring values that preserve data and control dependencies within a thread, i.e., in isolation (Unlike LC model)
    - Weak atomicity via correct ordering of load and stores within isolated sections
  - Java memory model semantics is preserved for volatile variables
  - Same semantics as Sequential Programs for data-race free programs

Side-Effects for Async-Escaping Methods

- **Async-Escaping Method Level Side-Effect (EMOD, EREF)**
  - **Sequential calls to methods that contain async constructs which are not wrapped in finish scopes**
  - **GMOD and GREF side-effects for async-escaping methods to be propagated in the call chain to their immediate enclosing finish (*IEF*)**

```
void foo () {
  async bar() // A
  … = p.x
  … = p.x
}

void bar () {
    p.x = …
}

void main () {
  p.x = …
  finish { // F
     foo ()
     … = p.x
  }
  … = p.x
 foo ()
}
```

GMOD (bar) = {p.x}

GMOD (A) = {p.x}

GMOD (foo) = {}

EMOD (foo) = {p.x}

EMOD (main) = {p.x}

Side-Effects for Finish Scopes

- **Finish Scope Level Side-Effect (FMOD, FREF)**
  - *Any async created within a finish scope scope must be completed before the statement after it is executed*
  - FMOD and FREF side effects comprise of the heap accesses for the asyncs within the finish scope

```
void foo () {
   async bar() // A
   … = p.x
   … = p.x
}

void bar () {
    p.x = …
}

void main () {
   p.x = …
   finish { // F
       foo ()
       ... = p.x
   }
   … = p.x
  foo ()
}
```

GMOD (bar) = {p.x}
GMOD (A) = {p.x}
GMOD (foo) = {}
EMOD (foo) = {p.x}
EMOD (main) = {p.x}

FMOD (F) = {p.x}
GMOD (main) = {p.x}

Side-Effect Analysis: putting all together

```
void main() {
  p.x = …
  s.w = …
  finish { //f
    async { //async_1
      if (…)  p.x = …
      isolated { q.y = …; … = q.y }
      … = p.x
    }
    … = p.x
    foo()
  }
  … = p.x
  … = s.w
}
```

```
void foo() {
    async bar() //async_2
    isolated { q.y = … }
    … = s.w
}
```

```
void bar() {
    r.z = …
    .. = r.z
}
```

AMOD = AREF = {*q.y*}

GMOD (bar) = GREF (bar) = {*r.z*}

GMOD (foo) = {}
GREF (foo) = {*s.w*}

EMOD (foo) = EREF (foo) = {*r.z*}

FMOD (f) = {*p.x, r.z*}
FREF (f) = {*p.x, r.z, s.w*}

GMOD (main) = GREF (main) = {*p.x, r.z, s.w*}

Example of Using Async-Finish to create a Parallel Loop

```
int iters = 0; delta = epsilon+1;
while ( delta > epsilon ) {
  finish {
    for ( jj = 1 ; jj <= n ; jj++ ) {
      final int j = jj;
      async { // finish-for-async can be replaced by foreach
        newA[j] = (oldA[j-1]+oldA[j+1])/2.0f ;
        diff[j] = Math.abs(newA[j]-oldA[j]);
      } // async
    } // for
  } // finish (join)
  delta = diff.sum(); iters++;
  temp = newA; newA = oldA; oldA = temp;
}
System.out.println("Iterations: " + iters);
```

Scalar Replacement for Load Elimination and Parallelism

- **Challenging to perform scalar replacement for load elimination transformation in the presence of parallel constructs**
  - **Interferences due to shared data accesses among parallel activities**
  - **Shared data accesses may not be properly synchronized**
    - **Compiler does not know if the input program is data-race free**
  - **Legality of the transformation depends on the underlying memory model supported by the programming language**
    - **Memory model determines the set of possible observable behaviors**
    - **It is desirable for a memory model to have same semantics for data-race free programs**

Example: Places to Co-locate Computation and Data

```
1) finish { // Inter-place parallelism
      final int x = … , y = … ;
      async (a) a.foo(x); // Execute at a's place
      async (b.distribution[i])
         b[i].bar(y); // Execute at b[i]'s place
   }

2) // Implicit and explicit versions of remote fetch-and-op
   a) a.x = foo(a.x, b.y) ;
   b) async (b) {
        final double v = b.y; // Can be any value type
        async (a) isolated a.x = foo(a.x, v);
      }
```

Scalar Replacement for Load Elimination

Parallelism-Aware Scalar Replacement Algorithm

- Compute side-effects for method calls and parallel constructs
  - Side-effects for async, finish scopes, and isolated blocks
- Append pseudo-defs and pseudo-uses to fields based on side-effects and isolation consistency memory model
- Create heap operands for field accesses including pseudo-defs and pseudo-uses
- Construct extended array-ssa form for the heap operands (handles both field accesses and array accesses)
- Perform global value numbering to compute Definitely-Same (*DS*) and Definitely-Different (*DD*) relations
- Perform data flow analysis to propagate value numbers for heap operands
- Eliminate loads if the value number is available

Reduction in Dynamic Field Accesses

FKS uses no side-effect analysis

Benchmark	# getfield original	#getfield after FKS Load elim.	#getfield after FKS+TRANS Load elim.	#getfield after PAR Load elim.	#getfield after PAR +TRANS Load elim.	Impr. relative to Original (%)	Impr. Relative to FKS	Impr. Relative to FKS+TRANS
CG-S	3.89E09	3.10E09	3.03E09	2.34E09	3.92E05	99.99%	99.99%	99.99%
MG-W	1.41E04	1.15E04	1.13E04	7.96E03	6.71E03	52.55%	41.72%	40.58%
MolDyn-B	1.19E10	7.91E09	5.82E09	4.91E09	3.11E09	73.89%	60.62%	46.49%
RayTracer-B	3.08E10	2.02E10	2.02E10	1.67E10	1.38E10	55.25%	31.93%	31.82%
Montecarlo-B	1.75E09	1.54E09	1.48E09	5.84E08	9.19E08	47.38%	40.48%	37.95%
specJBB-Java	1.19E09	1.02E09	8.95E08	6.65E08	5.78E08	51.56%	43.19%	35.43%

Decrease in dynamic counts of getfield operations of up to ~99.99%

Compilation-time Overhead

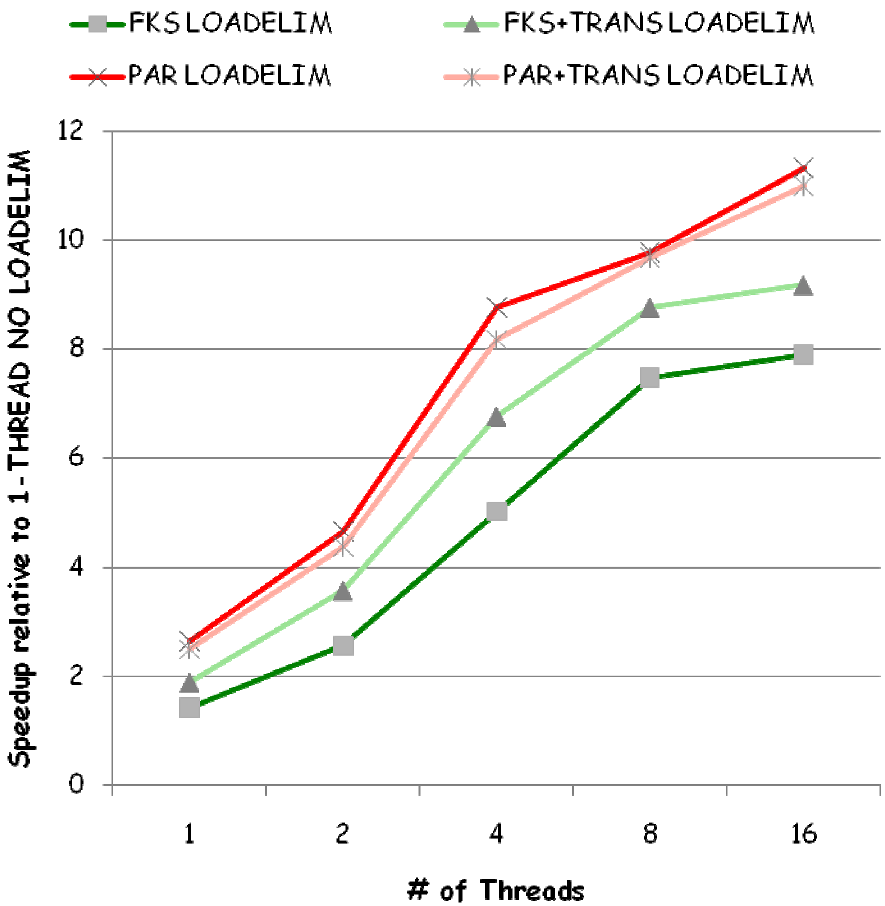
Benchmark	NO LOADELIM Total Comp time in ms	FKS LOADELIM ssa+loadelim time in ms	FKS LOADELIM TRANS time in ms	FKS LOADELIM Total Comp time in ms	PAR LOADELIM sideeffect time in ms	PAR LOADELIM ssa+loadelim time in ms	PAR LOADELIM TRANS time in ms	PAR LOADELIM Total Comp time in ms
CG-A	461	277	75	811	102	398	84	1137
MG-W	574	336	98	989	131	442	110	1348
MolDyn-B	263	194	35	493	76	255	47	673
RayTracer-B	275	157	35	468	77	246	44	670
Montecarlo-B	273	156	35	469	90	253	44	692
specJBB-JAVA	4336	1099	232	5625	580	1153	329	6867

Increase in compilation time for PAR LOADELIM in the range 1.22x to 1.47x compared to FKS+TRANS

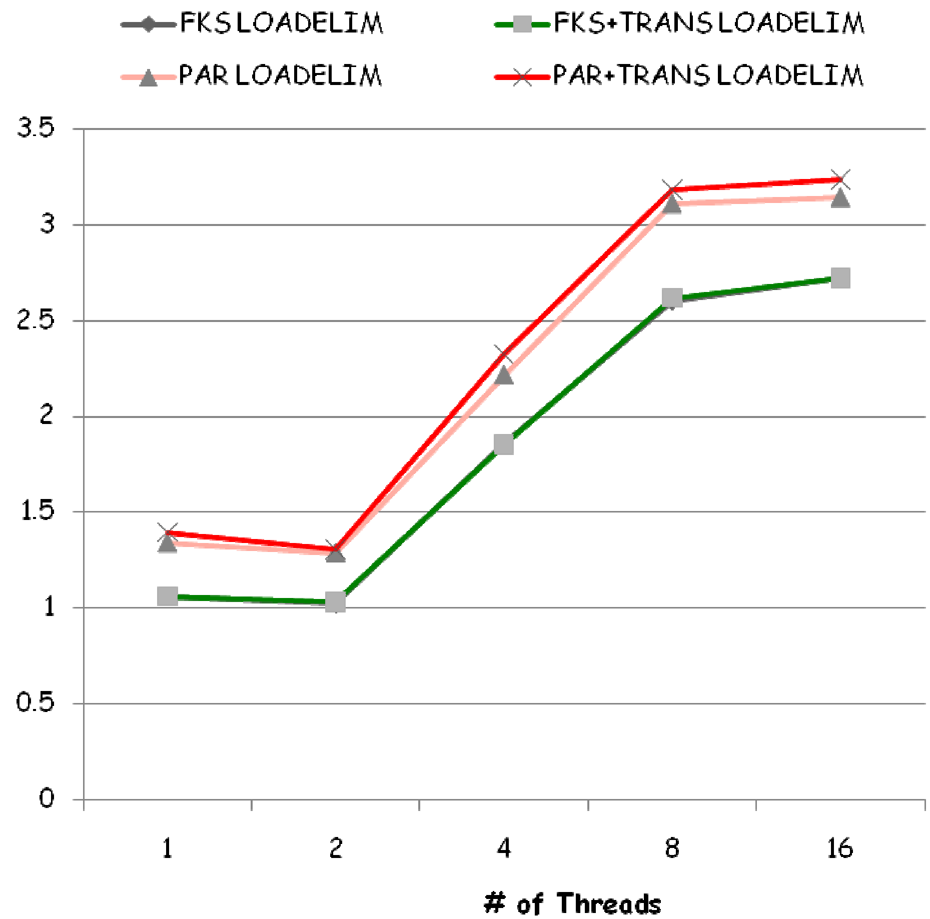
Benchmark Characteristics (static)

Benchmarks	async & foreach	finish	isolated
CG-A	5	5	0
MG-W	4	4	0
Moldyn-B	5	5	0
Raytracer-B	1	1	0
Montecarlo-B	1	1	0
specJBB-JAVA	1	1	169

Scalability on 4 Quadcore Intel Xeon

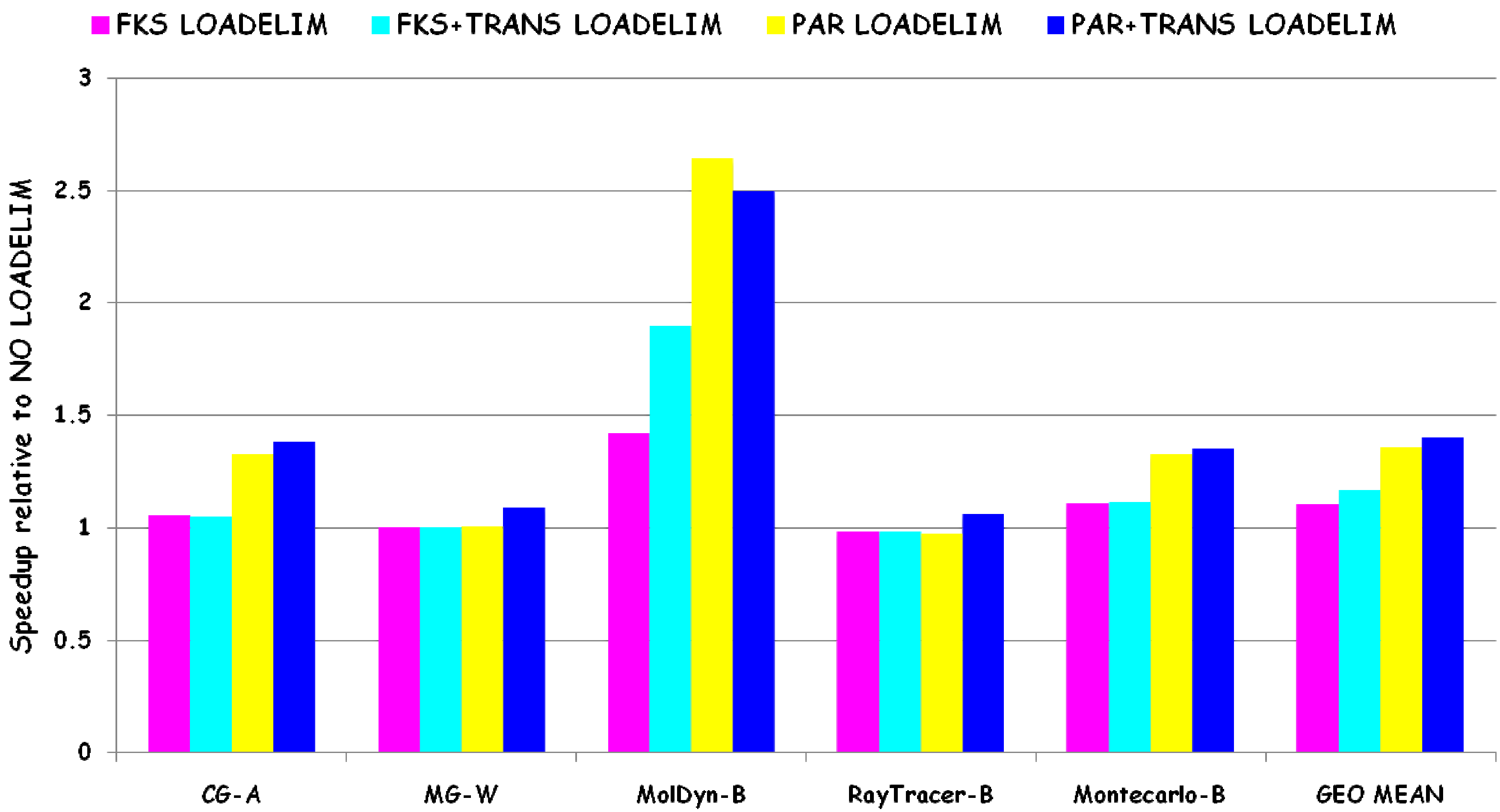


Moldyn



CG

Runtime Performance (1-Thread)



Speedup: up to 2.49x, and 1.48x on avg. compared to NO LOADELIM