MemoDyn: Exploiting Weakly Consistent Data Structures for Dynamic Parallel Memoization

Prakash Prabhu[2], Stephen R. Beard[1],
Sotiris Apostolakis[1], Ayal Zaks[3] and
David I. August[1]

[1]Princeton University [2]Google India [3]Intel Corporation

PACT 2018 – Session 3a – November 1, 2018

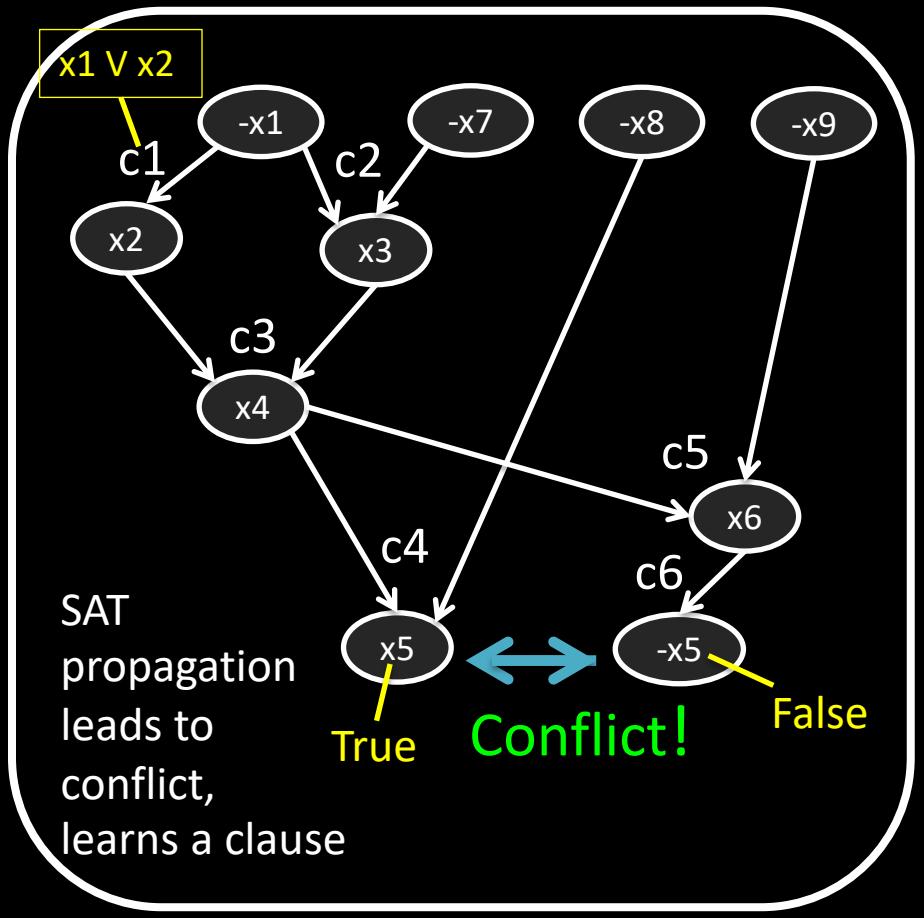
Why Implicit Parallel Programming?

	Explicit Parallel Programming	Automatic Parallelization	Implicit Parallel Programming
Ease of Sequential Programming	✗	✓	✓
Performance Portability	✗	✓	✓
Domain Knowledge	✓	✗	✓

Memoization

- Definition
  - Store intermediate results and retrieve them when needed again to avoid re-computation
- Usage
  - Mainly used to speed up convergence in combinatorial search and optimization problems
- Challenge?
  - Accesses to memoization data structures impose dependences that obstruct program parallelization
- Insight
  - Programs continue to function correctly even with weakly consistent memoization data structures
- Result
  - New parallelism opportunities

Motivating example: SAT solver

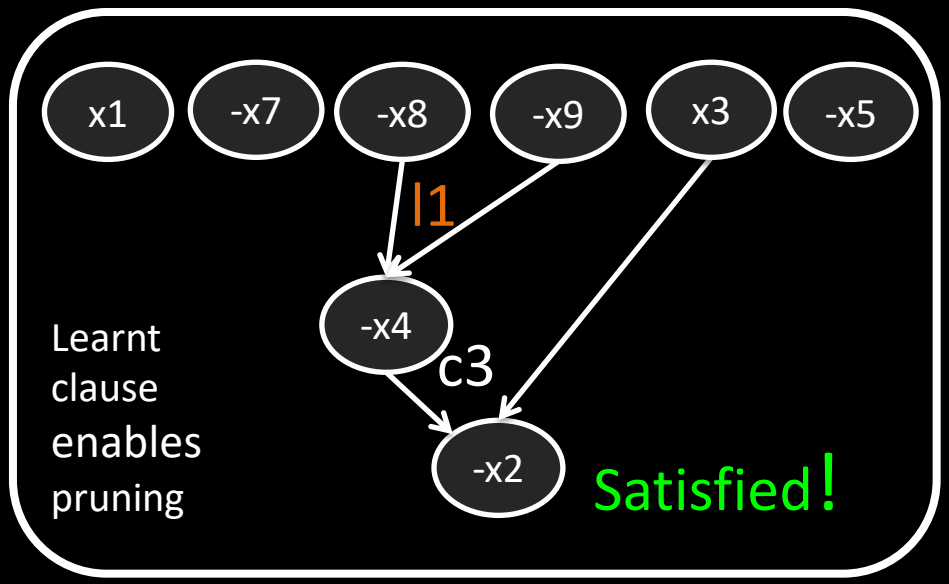


x1, …, xn : Boolean variables
c1, …, cn : Problem Clauses
l1, …, ln : Learnt Clauses

l1

Learnt Clause Data Structure memoizes l1

l1 = (-x4 V x8 V x9)



```
// SAT solver main loop
 while (1) {
    // Propagation
    for (int i=0; i < learnts.size(); ++i) {
      l = learnts[i];
      ...
    }
    if ( Conflict) {
      ... // Learn Conflict clause
      learnts.push(learnt_clause);
      ... // Backtrack
   } else  {
    // Reduce set of learnt clauses
    if (n >= nof_learnts) {
       for (int i=0; i < 2*n; ++i)
         learnts.remove(i);
      }
      if (model)
        return True;
      else
        ... // New variable decision
  }
}
```

Operations on learnt clause data structure in the search loop

Weaker Consistency Properties

Can execute out of order

A: learnts.push((-x4 ∨ x8 ∨ x9))

B: learnts.push((-x4 ∨ x8 ∨ -x6))

C: learnts.push((-x3 ∨ −x6))

D: learnts.push((-x1 ∨ -x10))

E: ~~learnts.remove(0)~~

F: for (i=0; i < learnts.size(); ++i) {
    clause = learnts[i];
    ...
}

Weak Deletion is safe

learnts

0	(-x4 ∨ x8 ∨ x9)
1	(-x4 ∨ x8 ∨ −x6)
2	(-x3 ∨ −x6)
3	~~(-x1 ∨ -x10)~~

Partial view is safe

Comparison of MemoDyn with Related Parallelization Frameworks

Programming Model	Properties		
	Out of Order	Partial View	Weak Deletion
N-way [1]	✓	✓	✗
Semantic Commutativity [2, 3,4]	✓	✗	✗
Galois [5]	✓	✗	✗
Concurrent Revisions [6], Cilk++ Hyperobjects [7]	✓	✓	✗
ALTER [8]	✓	✓	✗
This work	✓	✓	✓

[1] Cledat et al., “Efficiently speeding up sequential computation through the n-way programming model”, OOPSLA 2011
[2] Bridges et al., “Revisiting the Sequential Programming Model for Multi-Core”, MICRO 2007
[3] Prabhu et al., “Commutative Set: A Language Extension for Implicit Parallel Programming”, PLDI 2011
[4] Vandierendonck et al., “The Paralax Infrastructure: Automatic Parallelization With a Helping Hand”, PACT 2010
[5] Kulkarni et al., “Optimistic Parallelism requires abstractions”, PLDI 2007
[6] Burckhardt et al., “Concurrent programming with revisions and isolation types”, OOPSLA 2010
[7] Frigo et al., “Reducers and Other Cilk++ Hyperobjects”, SPAA 2009
[8] Udupa et al., “ALTER: Exploiting breakable dependences for parallelization”, PLDI 2011

Concurrent implementations of weakly consistent data structures

Prior Work

- Complete Sharing

Single shared data structure
Exploits OoO

- Privatization

Collection of private sequential data structures
Exploits OoO, Partial View

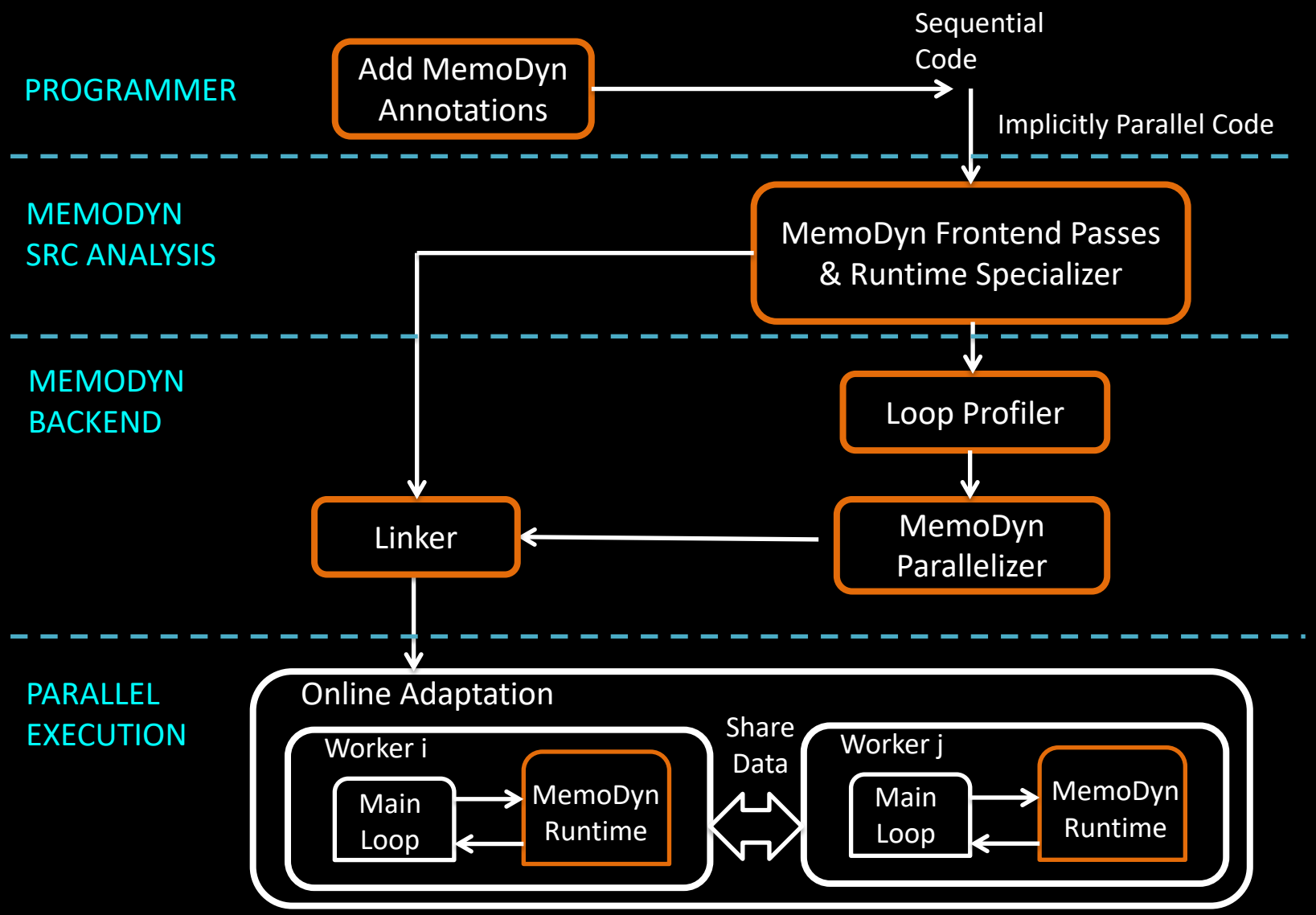
This Work

- Sparse Sharing

Static Adaptive

Hybrid that fully leverages weakened semantics for optimizing the tradeoff between consistency and parallelism.

MemoDyn Framework


PROGRAMMER
Add MemoDyn Annotations
Sequential Code
Implicitly Parallel Code
MEMODYN SRC ANALYSIS
MemoDyn Frontend Passes & Runtime Specializer
MEMODYN BACKEND
Loop Profiler
Linker
MemoDyn Parallelizer
PARALLEL EXECUTION
Online Adaptation
Worker i
Main Loop
MemoDyn Runtime
Share Data
Worker j
Main Loop
MemoDyn Runtime

```
#pragma MemoDyn(dstype, id)
      typed  obj;

dstype := SET | MAP | ALLOCATOR
```

MemoDyn data structure declarations

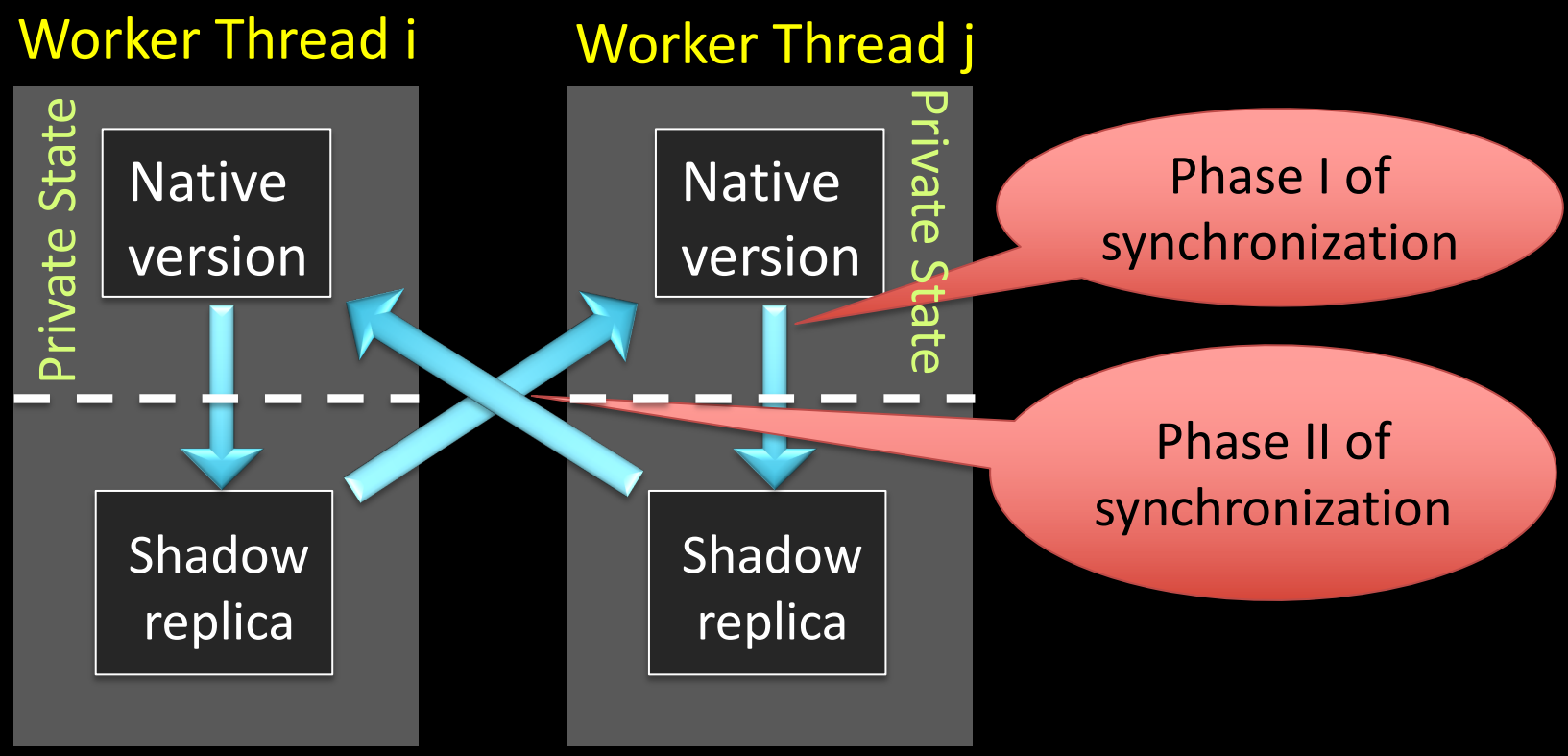
```
#pragma MemoDyn(id, itype)
      typer  class::fn(t1 p1, ...);

itype := INS | DEL | LU | ALLOC
          | SZ | DEALLOC | PROG | CMP
```

MemoDyn Interface declarations

```
class Solver {
   protected:
#pragma MemoDyn(SET, L)
      vec<Clause>  learnts;


      ...
      void attachClause   (Clause cr);
};

template<class T>
class vec {
   public:
#pragma MemoDyn(L, SZ)
      int size(void) const;
#pragma MemoDyn(L, LU)
      T& operator[](int index);
#pragma MemoDyn(L, INS)
      void push(const  T&elem);
#pragma MemoDyn(L, DEL)
      void remove(int i);
      ...
};

#pragma MemoDyn(L, PROG)
double Solver::progressEstimate() const {
   return (pow(F,i) * (end-beg)/nVars();
```

MemoDyn Runtime

- Synchronization
  - Two phases



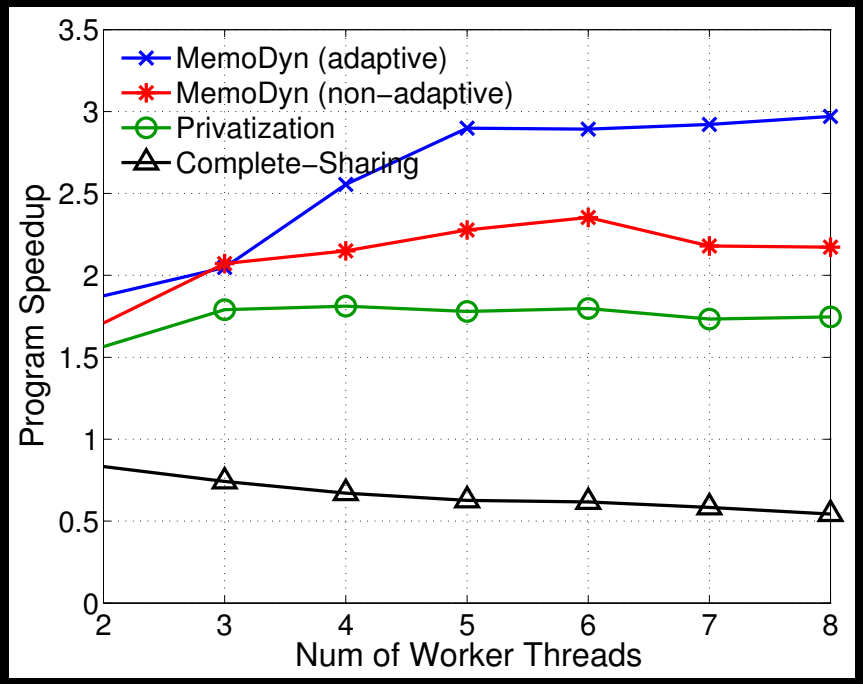
MemoDyn Runtime

- Tuning
  - Parameters:
    1. Save native version to local shadow replica (Phase 1) period
    2. Copy remote shadow replica to native version (Phase 2) period
    3. Sharing set size
  - Goal: Find the parameter configuration that maximizes the progress made by each worker
    - Seeded by profiling
    - One tuning thread per worker
    - Gradient ascent based

Evaluation

Evaluated:

- On 4 sequential programs
  - Minisat (SAT solver)
  - Genetic algorithm based graph optimization
  - Partial maximum satisfiability solver
  - Alpha-beta search based game engine
- On dual-socket quad core machine
- 4 parallelization schemes
  - MemoDyn adaptive
  - MemoDyn static
  - Privatization
  - Complete-sharing



Geomean speedup over 4 sequential programs

Conclusion

- Implicit parallel programming framework for parallelizing search loops with weakly consistent data structures
- Includes:
  - Language extensions for expressing weak semantics of data structures
  - Compiler-runtime system that leverages weak semantics for automatic parallelization and adaptive runtime optimization.
- Significant performance improvements over competing techniques
  - Minimal programmer effort

Thank you