MemoDyn: Exploiting Weakly Consistent Data Structures for Dynamic Parallel Memoization

Prakash Prabhu\textsuperscript{2}, Stephen R. Beard\textsuperscript{1}, \textbf{Sotiris Apostolakis}\textsuperscript{1}, Ayal Zaks\textsuperscript{3} and David I. August\textsuperscript{1}

\textsuperscript{1}Princeton University \quad \textsuperscript{2}Google India \quad \textsuperscript{3}Intel Corporation

PACT 2018 – Session 3a – November 1, 2018
## Why Implicit Parallel Programming?

<table>
<thead>
<tr>
<th></th>
<th>Explicit Parallel Programming</th>
<th>Automatic Parallelization</th>
<th>Implicit Parallel Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ease of Sequential Programming</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Performance Portability</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Domain Knowledge</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>
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

SAT propagation leads to conflict, learns a clause

Learnt Clause Data Structure memoizes l1
l1 = (¬x4 ∨ x8 ∨ x9)

Learnt clause enables pruning
Satisfied!
// 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
    }
}
Weaker Consistency Properties

A: `learnts.push([-x4 V x8 V x9])`
B: `learnts.push([-x4 V x8 V-x6])`
C: `learnts.push([-x3 V-x6])`
D: `learnts.push([-x1 V-x10])`
E: `learnts.remove(0)`
F: `for (i=0; i < learnts.size(); ++i) {
    clause = learnts[i];
    ...
}

Can execute out of order

Weak Deletion is safe

Partial view is safe
Comparison of MemoDyn with Related Parallelization Frameworks

<table>
<thead>
<tr>
<th>Programming Model</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Out of Order</td>
</tr>
<tr>
<td>N-way [1]</td>
<td>✓</td>
</tr>
<tr>
<td>Semantic Commutativity [2, 3,4]</td>
<td>✓</td>
</tr>
<tr>
<td>Galois [5]</td>
<td>✓</td>
</tr>
<tr>
<td>Concurrent Revisions [6], Cilk++ Hyperobjects [7]</td>
<td>✓</td>
</tr>
<tr>
<td>ALTER [8]</td>
<td>✓</td>
</tr>
<tr>
<td>This work</td>
<td>✓</td>
</tr>
</tbody>
</table>

[1] Cledat et al., “Efficiently speeding up sequential computation through the n-way programming model”, OOPSLA 2011
Concurrent implementations of weakly consistent data structures

Prior Work

- Complete Sharing
- Privatization

This Work

- Sparse Sharing
  - Static
  - Adaptive

Single shared data structure
Exploits OoO

Collection of private sequential data structures
Exploits OoO, Partial View

Hybrid that fully leverages weakened semantics for optimizing the tradeoff between consistency and parallelism.
MemoDyn Framework
Add MemoDyn Annotations

Sequential Code

Implicitly Parallel Code

MemoDyn Frontend Passes & Runtime Specializer

Loop Profiler

MemoDyn Parallelizer

Linker

Online Adaptation

Worker i

Main Loop

MemoDyn Runtime

Worker j

Main Loop

MemoDyn Runtime

Share Data
### MemoDyn data structure declarations

```cpp
#pragma MemoDyn(dstype, id) typed obj;
dstype := SET | MAP | ALLOCATOR
```

### MemoDyn Interface declarations

```cpp
#pragma MemoDyn(id, itype) typer class::fn(t1 p1, ...);
iday := INS | DEL | LU | ALLOC
    | SZ | DEALLOC | PROG | CMP
```

```cpp
class Solver {
    protected:
#pragma MemoDyn(SET, L)
    vec<Clause> learnts;
    ...
#pragma MemoDyn(SZ)
    void attachClause (Clause cr);
};
```

```cpp
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);
    ...
};
```

```cpp
#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
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