MemoDyn: Exploiting Weakly Consistent Data Structures for Dynamic Parallel Memoization

Prakash Prabhu*
Google India
prakashp@google.com

Stephen R. Beard
Princeton University
sbeard@princeton.edu

Sotiris Apostolakis
Princeton University
sapostolakis@princeton.edu

Ayal Zaks
Intel Corporation, Haifa, Israel
ayal.zaks@intel.com

David I. August
Princeton University
august@princeton.edu

Abstract

Several classes of algorithms for combinatorial search and optimization problems employ memoization data structures to speed up their serial convergence. However, accesses to these data structures impose dependences that obstruct program parallelization. Such programs often continue to function correctly even when queries into these data structures return a partial view of their contents. Weakening the consistency of these data structures can unleash new parallelism opportunities, potentially at the cost of additional computation. These opportunities must, therefore, be carefully exploited for overall speedup. This paper presents MemoDyn, a framework for parallelizing loops that access data structures with weakly consistent semantics. MemoDyn provides programming abstractions to express weak semantics, and consists of a parallelizing compiler and a runtime system that automatically and adaptively exploit the semantics for optimized parallel execution. Evaluation of MemoDyn shows that it achieves efficient parallelization, providing significant improvements over competing techniques in terms of both runtime performance and solution quality.

CCS Concepts

• **Software and its engineering → Concurrent programming structures**; **Compilers**; **Runtime environments**;

Keywords

Implicit parallelism, programming model, weakly consistent data structures, automatic parallelization, memoization, runtime, adaptivity, tuning

*Work performed at Princeton University.

PACT '18, November 1–4, 2018, Limassol, Cyprus

ACM ISBN 978-1-4503-5986-3/18/11…$15.00
https://doi.org/10.1145/3243176.3243193

ACM Reference Format:

Prakash Prabhu, Stephen R. Beard, Sotiris Apostolakis, Ayal Zaks, and David I. August. 2018. MemoDyn: Exploiting Weakly Consistent Data Structures for Dynamic Parallel Memoization. In *International conference on Parallel Architectures and Compilation Techniques (PACT '18), November 1–4, 2018, Limassol, Cyprus.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3243176.3243193

1 Introduction

Search and optimization problems play an important role in many modern-day desktop and scientific applications. These problems are typically combinatorial in nature, having search spaces that are prohibitively expensive to explore exhaustively. As a result, many real-world algorithmic implementations for these problems employ various optimization techniques. One well known and common technique is memoization [31]: as the search space is explored, new facts are recorded; these facts are later retrieved to prune or guide the search. Examples of data structures for memoization include kernel caches in support vector machines [23], learned clause databases in SAT solvers [15], elite sets in genetic algorithms [30], cut pools in MILP solvers [29], and transposition tables in chess programming [41]. Regardless of what they are called in each specific domain, they all have the same basic memoization property: trade-off space for computation time by remembering previously computed results. This optimization technique greatly improves the running times of such search algorithms without affecting their correctness. There are even tools to help programmers identify memoization opportunities in their programs [11].

The emergence of multicore into mainstream computing presents a tremendous opportunity to parallelize many search and optimization algorithms. Explicitly parallelizing search loops by synchronizing accesses to the memoization data structure is one way of optimizing such programs. This is, however, a slow, manual, and error-prone process that results in performance-unportable and often sub-optimal programs.

Prior work demonstrates how high-level semantic programming extensions can facilitate automatic parallelization. Semantic commutativity extensions [7, 27, 37] allow functions that atomically access shared state to execute out-of-order concurrently, breaking flow

dependences across function invocations. Cilk++ hyperobjects [17] and the N-way programming model [9] provide programming support to implicitly privatize data for each parallel task. ALTER [46] presents an optimistic execution model allowing parallel tasks to read stale values of shared data as long as they do not write to the same locations.

Programs accessing memoization data structures can be parallelized by the methods described above, but each has its pitfalls: synchronizing *every* access to a shared data structure as implied by semantic commutativity is expensive, given the high frequency of access to these data structures; complete privatization without communicating any memoization data can adversely affect the convergence times of search loops by providing fewer pruning opportunities; and with the ALTER model, conflicts based on writes to memory would frequently occur due to updates to memoization data structures.

This work is based on the following insight: programs that access memoization data structures continue to function correctly even when queries to the data structure return a partial view of its contents, or when deletions are not persistent. This weaker consistency requirement can be leveraged to break dependences between data structure operations, facilitating automatic parallelization of the main search loops. Once parallelized, it is important to optimize the trade-off between increased parallelism and potential increase in computation. Frequent communication of memoization data between parallel tasks can reduce required computation at the cost of synchronization overheads, and conversely little or no communication of memoization data may increase required computation while reducing synchronization.

This paper presents MemoDyn, a framework for efficiently parallelizing search loops that access memoization data structures. MemoDyn consists of,

- semantic language extensions for weakly consistent data structures, which expose parallelism in search loops using them;
- a compiler that automatically parallelizes search loops; and,
- a runtime system that adaptively optimizes parallel configurations of weakly consistent data structures.

Table 1 compares MemoDyn with frameworks that leverage relaxed semantics for parallelization. The main advantage of MemoDyn stems from expressing much weaker semantics than any of the other frameworks, and having the MemoDyn parallelization framework leverage the weaker semantics to realize an adaptively synchronized parallel scheme as opposed to static parallel schemes realized by others.

2 Motivation

MemoDyn is motivated by the Boolean satisfiability problem (SAT), a well-known search problem with many important applications. A SAT solver attempts to solve a given Boolean formula by assigning values to variables such that all clauses are satisfied, or determine that no such assignment exists. Figure 1 shows main parts of a SAT solver's [15] C++ implementation.

The main search loop of a SAT solver selects an unassigned variable, sets its value, and recursively propagates this assignment until either all variables are assigned (thereby completing a solution) or a constraint becomes conflicting under the current assignment. In case of a conflict, an analysis procedure learns a clause implying the conflict, records it in a memoization data structure (Line 11), and applies backtracking.

Clause learning greatly helps subsequent iterations prune their search space. Disabling clause learning led to an average slowdown of 8.15x when running a sequential SAT solver across several workloads. However, while more learning implies more pruning opportunities, traversals of the memoization data structure (Line 5) can slow down if its size grows excessively, hence the SAT solver loop periodically removes "useless" learned clauses (Line 21).

Considering the intractable nature of the SAT problem, parallelizing such loops has the potential to drastically reduce search times. One common approach to parallelize search and optimization algorithms is *multisearch* [44]. In this approach, multiple parallel workers search in distinct regions of the search space until some worker finds a solution. Convergence times are greatly improved when clauses learned by each worker are shared [20]. However, manually parallelizing a SAT search loop using explicit parallel programming requires considerable effort. Tools for automatic parallelization can relieve programmers of this effort but are constrained by the need to respect sequential semantics. In this example, reads, inserts, and deletes into the `learnts` data structure within the search loop are inter-dependent (Lines 5, 11 and 21), inhibiting automatic parallelization.

The use of `learnts` data structure in the context of SAT solvers, however, requires much weaker consistency than imposed by a sequential programming model. In particular, the SAT search loop will function correctly even when reads from `learnts` only return a partial set of clauses inserted so far or when deleted clauses persist. This weaker semantics of `learnts` can be used to break dependences between data structure operations and facilitate automatic parallelization. Programming extensions can express this weaker semantics, enabling transformation tools to parallelize the loop without sacrificing ease of sequential programming. Note that a SAT solver using `learnts` with weaker consistency property may explore the search space differently from a sequential search, which may result in a different satisfying assignment.

Prior work on semantic extensions to the sequential programming model for parallelism impose much stronger requirements on the `learnts` data structure operations than required by SAT solvers. In these solutions, all inserts into `learnts` have to eventually succeed [46] and operations either access one shared `learnt` data structure which is always synchronized [7, 27, 37] or operate on multiple private copies with no clause sharing [9, 17]. Choosing "all" or "nothing" synchronization strategies for `learnts` data structure can adversely affect parallel performance: using fine-grain synchronization mechanisms to completely share every clause learned amongst the workers can lead to increased overheads, while having private copies with no sharing can increase the convergence time due to redundant computation.

Programming Model	Concept (Section 3)				Specific Parallel Implementation (Sections 4 and 5)		
	Out of Order	Partial View	Weak Deletion	Annotation Type	Parallelization Driver	Sharing	Synchronization
N-way [9]	✓	✓	×	I	Runtime	None	Static
Semantic Commutativity [37]	✓	×	×	I	Compiler	All	Static
Galois [27]	✓	×	×	I	Runtime	All	Static
Cilk++ hyperobjects [17]	✓	×	×	E	Programmer	None	Static
ALTER [46]	✓	✓	×	I	Compiler	All	Static
MemoDyn [this paper]	✓	✓	✓	I	Compiler & Runtime	Sparse	Static/Adaptive

Table 1: Comparison between MemoDyn and related parallelization frameworks (I: Implicitly Parallel, E: Explicitly Parallel)

```
1  lbool Solver::search(int nof_learnts) {
2   model.clear();
3   while (1) {
4    // propagate() accesses learnts[i]
5    Constr confl = propagate();
6    if (confl != NULL) { // Conflict
7     if (decisionLevel() == root_level)
8       return False;
9     learnt_clause =analyze(confl, backtrack_level);
10    cancelUntil(max(backtrack_level, root_level));
11    learnts.push(learnt_clause);
12    decayActivities ();
13    ... // backtrack
14   }
15   else { // No conflict
16    ...
17    if (learnts.size()-nAssigns() >= max_learnts){
18      // Reduce the set of learnt clauses
19      sort(learnts);
20      for (int i=0; i < learnts.size()/2; ++i)
21        learnts.remove(learnts[i]);
22    }
23    if (nAssigns () == nVars ()) {
24      // Model found:
25      model.growTo(nVars ());
26      ...
27      cancelUntil (root_level);
28      return True;
29    } else {
30      // New variable decision
31      lit p = lit (order.select());
32      assume (p);
33    }
34   }
35  }
36 }
```

```
37 class Solver {
38   protected:
39 #pragma MemoDyn(SET, L)
40     vec<CRef> learnts;
41 #pragma MemoDyn(ALLOCATOR, A)
42     ClauseAllocator ca;
43     ...
44     void attachClause     (CRef cr);
45 };
46
47 template<class T> class vec {
48   public:
49 #pragma MemoDynI(L, SZ);
50     int size(void) const;
51 #pragma MemoDynI(L, ILU)
52     T& operator [](int index);
53 #pragma MemoDynI(L, INS)
54     void push(const T& elem);
55 #pragma MemoDynI(L, DEL)
56     void remove(const T& elem);
57     ...
58 };
59
60 class ClauseAllocator : public RegionAllocator
61 {
62   public:
63 #pragma MemoDynI(A, ALLOC)
64     void reloc(CRef& cr, ClauseAllocator& to);
65 #pragma MemoDynI(A, DEALLOC)
66     void free(CRef cid);
67     ...
68 };
71 #pragma MemoDynI(L, PROG)
72 double Solver::progressEstimate() const {
73     return  (pow(F,i)* (end-beg))/ nVars();
74 }
```

Figure 1: Motivating Code Example (a) SAT main search loop with learning (b) SAT solver declarations with MemoDyn annotations

Instead, this paper introduces a hybrid synchronization strategy named *sparse sharing* that completely exploits the weak consistency property of `learnts` in a way that has much better performance characteristics than private and complete sharing strategies. In sparse sharing, each worker maintains a copy of the `learnts` data structure and *selectively synchronizes* with other workers by exchanging *subsets* of locally learned clauses at certain intervals, balancing the trade-off between synchronization overhead and required computation.

The running times of the SAT solver, similar to many search and optimization algorithms, varies widely depending on characteristics of its input. Quite often, it exhibits phase changes in its computation [47]. In such scenarios adapting the synchronization strategy to the differing progress rates of parallel workers (measured on Line 72 in our example) has the potential to substantially accelerate search convergence. For instance, during certain phases some workers might quickly learn facts that could help other workers prune their search space considerably. Online adaptation tunes to this varying runtime behavior, effectively enabling workers to cooperatively find a solution faster than otherwise possible.

Varying the synchronization strategy is only one parameter exposed by the weakened semantics of the `learnts` data structure. Allowing inserted clauses to disappear opens new opportunities for efficient eviction mechanisms, and mechanisms for deciding which clauses to share, with whom and when. These parameters are important due to the tradeoff between the usefulness of the `learnts` database and the time spent synchronizing it, which is proportional to its size. Given the dynamic nature of SAT search, adapting these parameters too at runtime in response to the varying progress made by different workers can greatly improve convergence times. Adaptive sparse sharing results in a speedup of 5.2x on eight cores, a gain of 148% over earlier schemes (Section 6).

3 Weakly Consistent Data Structures

3.1 Semantics

We describe the semantics of a weakly consistent data structure in terms of operations supported on the data structure through its interface. A *weakly consistent set*, for example, is defined as an abstract data type that stores values, in no particular order, and may contain repeated values similar to multiset. The semantics of the operations supported by a weakly consistent set is as follows:

(1) **Out of order mutation:** The insertion operations of a weakly consistent set can be executed in a different order from that of a sequential specification. Similarly, deletions of multiple elements can execute out of order with respect to each other and also with respect to earlier insertions of non-identical elements. Both these properties follow directly from a conventional set definition [22].

(2) **Partial View:** The set supports lookup operations that may return only a subset of its contents, giving only a partial view of the data structure that may not reflect the latest insertions. Each element returned by a lookup must, however, have been inserted at *some earlier* point in the execution of the program.[1] This partial view semantics corresponds to the concept of *weak references* [13] in managed languages like Java, where it is used for non-deterministic eviction of inserted elements to enable early garbage collection. In the SAT example, a partial view of the learned clause set during propagation may only reduce pruning opportunities, but has no effect on program correctness.

(3) **Weak Deletion:** Deletions from a weakly consistent data structure need not be persistent, with the effect of deletions lasting for a non-deterministic length of time. When combined with partial view, this semantic implies that between two lookup operations that are invoked after a deletion, certain elements not present in the first lookup may appear during the second lookup without any explicit insertion operation. In the SAT example, the weak deletion property safely applies to the learned clause set due to the temporally-agnostic nature of learned clauses: Since a learned clause remains globally true regardless of when it was learned, it is safe to re-materialize deleted clauses.

The abstract semantics outlined above translates into concurrent semantics by adding one additional property: atomicity of each operation. A weakly consistent set can have different concurrent implementations, each trading consistency for different degrees of parallelism. We describe three realizations of a weakly consistent set, two of which – privatization and complete sharing have been studied in the context of other parallel programming models. The third realization is sparse sharing, which is introduced in this paper and implemented by MemoDyn. Sparse sharing takes full advantage of weakened semantics for optimizing the tradeoff between consistency and parallelism. In the following, the term *client* refers to an independent execution context such as a thread or an OS process.

[1]The terms *later* and *earlier* relate to a time-ordered sequence of operations invoked on the data structure as part of an execution history or trace.

(1) **Privatization** [9, 17]: A weakly consistent set can be implemented as a collection of multiple private replicas of a sequential data structure, one for each client. Methods invoked by a client operate on its private replica. There is no interference between clients, and all operations are inherently atomic without any synchronization. Insertion and deletion operations on a single replica execute in sequential order, while they execute out of order with respect to other replica operations. In any time-ordered history of operations, lookups in a client only return elements inserted earlier into a local replica and not the remote ones, thus giving only a partial global view of the data structure. However, deletions are persistent: once an element is deleted in a replica, it will not be returned by a subsequent lookup on that replica without an explicit insertion of the same element.

(2) **Complete Sharing** [7, 27, 37]: A weakly consistent set is realized by a single data structure that is shared amongst all clients. Operations performed on the data structure by clients are made atomic by wrapping them in critical sections and employing synchronization. This realization is implied by and implemented via semantic commutativity annotations on the operations of the data structure. In this method insertions and deletions of non-identical elements can occur out of order. Lookups present a complete view of the data structure for each client and return the most up-to-date, globally available contents of the set. Deletion of an element is persistent, with effects immediately visible across all clients.

(3) **Sparse Sharing**: A third realization of a weakly consistent set combines the above two to take full advantage of the partial view and weak deletion property. This scheme uses a collection of replicas, one for each client, and non-deterministically switches between two sharing modes. In private mode, each operation is performed on the local replica without any synchronization. In sharing mode, insertions are propagated transparently to remote replicas while deletions continue to apply to the local replica only. In this model, lookups return a partial view of cumulative contents of all replicas: elements made visible to a client correspond to a mix of insertions performed locally and remotely at some earlier point in execution. Deletions are weak: an element deleted in a local replica can reappear if it has been propagated to a remote replica prior to deletion and is propagated back to the local replica transparently at some point after deletion. Note that even with weak deletion, infrequently used clauses will be completely removed by the MemoDyn runtime (Section 5), first from local replicas and eventually globally if they are not useful to any client.

The weak consistency concept easily extends from sets to other sequential data structures. In particular, a *weakly consistent map* is very useful for applications performing associative lookups of memoized data. The use of kernel caches in SVMs and transposition tables in alpha-beta search [41] are instances of such weakly consistent maps.

MemoDyn Data Structure Declarations	`#pragma MemoDyn(dstype, id)`
	`  `$type_d$` obj;`
	
	`dstype := SET | MAP | ALLOCATOR`

MemoDyn Interface Declarations	`#pragma MemoDynI(id, itype)`
	`  `$type_r$` class::fn(`t_1` `p_1`, ...);`
	
	`itype := INS | DEL | ILU | ALLOC`
	`  SZ | DEALLOC | PROG | CMP`

Figure 2: MemoDyn **Syntax**

3.2 Syntax

This paper describes a MemoDyn data structure using a standard abstract interface. This interface corresponds to that of a set or a map class with member functions for insertion, deletion, indexed search and querying the data structure's size.

MemoDyn extensions are specified as `pragma` directives within a client's sequential program.[2] `pragma`s are used for two reasons: first, arbitrary data structure implementations within a client can be regarded as weakly consistent by annotating relevant accessor and mutator interfaces using MemoDyn's `pragma`s without major re-factoring. Second, the use of `pragma`s preserves the sequential semantics of the program when the annotations are elided, and allows programs with MemoDyn's annotations to be compiled unmodified by C++ compilers that are unaware of MemoDyn semantics.

Figure 2 shows the syntax of MemoDyn directives which include two parts: extensions to annotate a data structure with a MemoDyn type at its instantiation site, and for declaring the data structure's accessors and mutators as part of the MemoDyn interface. The two main MemoDyn data structures are the weakly consistent set (SET) and map (MAP), with corresponding member functions for insertion (INS), deletion (DEL), indexed lookup (ILU), and querying the size of the data structure (SZ). Combining ILU and SZ gives a multi-element lookup functionality. An allocator (ALLOCATOR) class can be optionally associated with a weakly consistent data structure. This class is akin to allocators for C++ standard template library classes, and can be used when collections of pointers that rely on custom memory allocation are annotated as weakly consistent. The associated allocation (ALLOC) and deallocation (DEALLOC) methods are used by MemoDyn to transparently orchestrate replication and sharing among parallel workers. Finally, the parent solver class containing a weakly consistent data structure can specify a member function that reports search progress (PROG) – a real valued measure between 0 and 1, and a member function to compare the relative quality of weakly consistent element pairs (CMP). Both these functions serve as hints to the MemoDyn runtime for optimizing sparse sharing. Figure 1b shows MemoDyn directives applied to the SAT example. Note that the specific signature types of the annotated methods do not matter and thus they do not need to match the ones in Figure 1b.

Target data structures may lack certain of these member functions. For example, inherited functions are implicitly declared (e.g. copy constructors) and a function that reports search progress may not be implemented. For the evaluated programs, the necessary source code changes for compliance with MemoDyn's abstract interface are discussed throughout section 6 and are summarized in Table 2.

[2] MemoDyn also provides a library of weakly consistent data structures based on C++ templates akin to STL that can be directly used.

4 The MemoDyn Compiler

In the MemoDyn parallelization model, the programmer first annotates a sequential program using `pragma`s at interface declarations and data structure instantiations. This program is fed into the MemoDyn compiler which analyzes and extracts semantic information in its frontend and later introduces parallelism in its backend. The resulting parallelized program executes in conjunction with the MemoDyn runtime that performs online adaptation.

Frontend. The frontend is based on clang++ [28]. It includes a `pragma` parser, a type collector, and a source-to-source rewriter. The `pragma` parser records source level and abstract syntax tree (AST) level information about weakly consistent data structures and interface declarations. The type collector module uses this information to correctly deduce fully qualified source level type information of weakly consistent containers and checks that the type information conforms to expected prototypes. The rewriter module adds and initializes additional data member fields into the parent class of weakly consistent types. In particular, it creates a unique identifier field for distinguishing between different object instances belonging to different parallel workers at runtime, to support an object based parallel execution model [24] within MemoDyn.

The output of the rewriter is translated by the frontend from C++ code into LLVM IR [28]. Meta-data describing MemoDyn annotations is embedded within this IR destined for the backend. The frontend also creates runtime hooks to: (a) instantiate concrete types for abstract data types declared within the MemoDyn runtime library; (b) synthesize functors for measuring the progress of each solver; and within (c) specialize the runtime's function templates with the concrete types and interface declarations of the client program. The outcome of these steps is the creation of a version of the MemoDyn runtime library specialized to the current client program.

Backend. The backend first profiles the LLVM IR to identify hot loops. Once an outermost hot loop is identified, it injects parallelism into the IR by transforming code to spawn threads at startup, to replicate parent solver objects for granting ownership of each replica to a newly spawned thread, and to start parallel execution. The search loop is then augmented with calls into the runtime for sparse sharing. These include calls to save a parallel worker's native weakly consistent set for exchange with other workers, and calls to exchange weakly consistent data between workers at periods determined by the runtime. The final step in parallelizing the loop handles graceful termination — ensuring cancellation of remaining workers when one worker completes execution. MemoDyn follows the *cooperative cancellation* model [42] where the compiler inserts cancellation checks at well-defined syntactic points in the code. In our implementation, these points correspond to the return sites of the transitively determined callers to runtime functions. Finally,

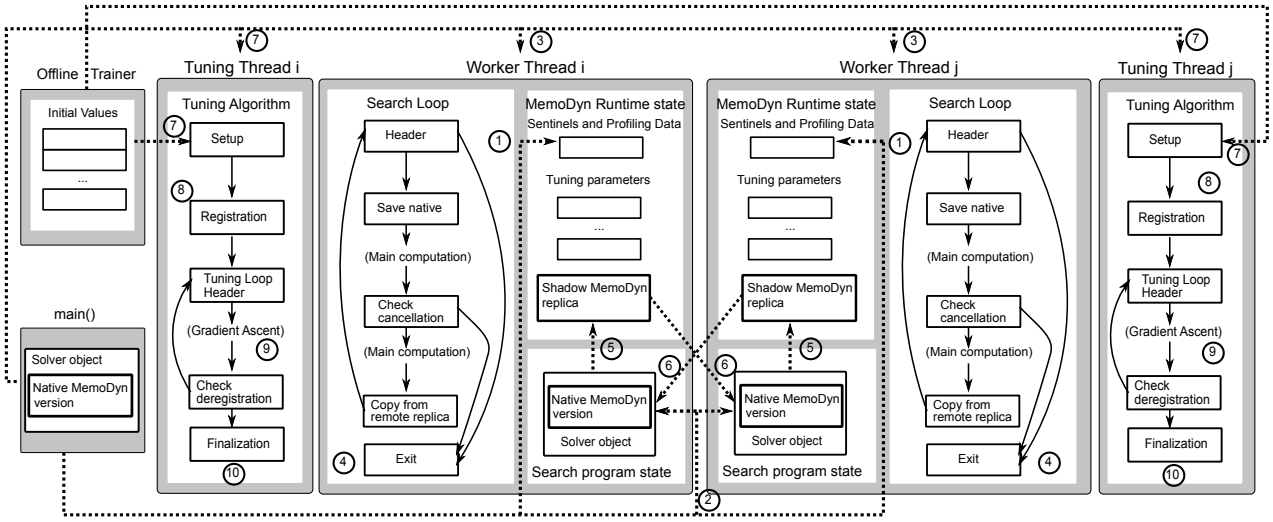


Figure 3: The MemoDyn runtime execution model

the parallelized IR is linked with the specialized MemoDyn runtime library to generate a parallel executable.

5 The MemoDyn Runtime

5.1 Implementation

The MemoDyn runtime maintains a clear separation between its parallelization, synchronization, and tuning subsystems. The runtime interface is based on generic data types, makes no references to C++ STL or any specific parallel libraries and hence can be targeted by both compilers and programmers alike. The current implementation of the MemoDyn runtime is based on POSIX-threads. Figure 3 shows a schematic of MemoDyn's parallel runtime execution.

The Parallelization subsystem controls parallelism injection, management, and termination as follows:

- **Initialization (1), Search Object Replication (2), and Parallel Worker Creation (3)**: The MemoDyn runtime first initializes bookkeeping and profiling data; creates a one-to-one map between parallel workers and replicated search objects based on an object-based concurrency model [24]. The replication function uses a reference to the original solver object from the sequential program to create new solver objects within the runtime. Additionally, functors encapsulating (a) the main driver that performs the search and (b) a progress measure function for online adaptation are recorded. Next, MemoDyn creates independent threads of control, each given ownership of its unique search object, initialized to search from a distinct point in the search space. State separation between parallel workers via object based replication in MemoDyn ensures safe automatic parallelization.
- **Finalization (4)**: On successful completion, a worker sets a global cancellation sentinel which is polled by all workers at appropriate cancellation points. The other workers thus quickly exit their search and terminate gracefully.

The Synchronization subsystem orchestrates sparse sharing in two phases: in the first phase, data from a worker's *native* weakly consistent data structure is copied into a *shadow replica* that resides within the MemoDyn runtime. In the second phase, remote workers pull data from other workers' shadow replica into their own native version.

Sparse sharing was designed according to this two-phased protocol for several reasons. First, a shadow replica distinct from a native version allows remote workers to exchange data asynchronously with a more efficient coarser grained synchronization mechanism. The data exchange is decoupled from a local worker's operations on its native version, thereby preventing incorrect program behavior due to data races. Second, two-phased sharing has the effect of buffering: given a sequence of insert and delete operations on a native version, only its net result is copied into the shadow replica which amortizes the synchronization costs over this sequence. Third, this protocol naturally leads to a sharing discipline that avoids deadlocks. In particular, no remote worker ever requests access to a worker's shadow replica while holding access to another worker's shadow replica and so hold-and-wait conditions necessary for a deadlock do not arise. Sparse sharing has these steps:

- **Saving into shadow replica (5)**: Given a native version of a weakly consistent set, iterates through the elements of the set adding them to the shadow replica. If the shadow replica is full, an existing element is chosen and replaced using a comparison function designated by MemoDyn annotations. Updates to the shadow replica are performed within a critical section to prevent remote workers from accessing the replica's intermediate state.
- **Copying from remote shadow replica (6)**: Copies data from a remote worker's shadow replica into its own native version. This function is invoked asynchronously by each worker. Interference may arise only when two or more workers attempt to copy data from the same shadow replica. Each worker attempts to copy data by acquiring exclusive access to a shadow replica in sequence,

with the order of acquisition and release determined a priori. As above, existing elements in a native version are replaced if needed using a comparison function.

The synchronization interface includes an optional allocator object to support custom memory management of weakly consistent data structures. In such a case, an instance of this allocator class is used within the MemoDyn runtime to manage a shadow replica's memory allocation and deallocation (see `minisat`, Section 6.1).

The Tuning subsystem implements MemoDyn's online adaptation and allows for both synchronous [43] and asynchronous models of tuning [45]. Synchronous tuning is within a worker's thread of computation, while asynchronous tuning is done concurrently by tuner threads that update tunable program state within a worker periodically. The MemoDyn library maintains a clear separation between tuning algorithm state and the search/parallel program state. Tuning works as follows:

- **Setup (7), Registration (8)**: Initializes tuning algorithm state, including seed values for tuning parameters from the environment. For asynchronous tuning, lightweight tuning threads are created as part of the setup – one for each parallel worker. Tuning variables along with lower and upper bounds for these variables are recorded to constrain tuning to within these limits.
- **Tuning (9)**: Parameter tuning for each solver is done independently of each other. Each tuner perturbs variables registered for tuning, observes its effects by estimating change in progress using the progress measure functor recorded internally within MemoDyn, and then uses it for optimization.
- **Deregistration and Finalization (10)**: Parameters can be deregistered on the fly to disable tuning during certain phases. A tuning algorithm terminates when a worker completes its search. In asynchronous tuning, pre-emptive thread cancellation primitives terminate tuning threads.

5.2 Online adaptation

Desired Properties. Online adaptation within MemoDyn poses a number of interesting challenges. First, concurrent access to a search program's runtime state by both a worker and a tuning thread (in asynchronous tuning) during execution can result in inconsistencies if mutual exclusion is not ensured, or in performance penalties otherwise. In MemoDyn, parameters tied to shadow replicas are tuned within the runtime rather than those of a native version. Moreover, a search thread has limited access to a shadow replica: only during phases of synchronization that are explicitly controlled by the runtime. Second, online adaptation is useful only when programs run sufficiently long for a tuner to sample enough parameter configurations to optimize at runtime. Third, tuning algorithms should be lightweight and should not interfere with or slow down the main search algorithm computation. This implies that a tuning algorithm should either be invoked infrequently or involve only relatively inexpensive computations, and should not increase contention for hardware resources shared with the search threads. The tuning algorithm used within MemoDyn has these properties.

Approach. The goal of online adaptation in MemoDyn is to improve the overall execution time of parallel search. Since search programs typically contain one main loop that is iterated many times and do not exhibit regular memory access or predictable control flow patterns which can be monitored, we rely on application level progress measures that serve as a proxy for online performance. Most solvers already provide such a measure, expressable using a MemoDyn annotation.

MemoDyn tuning strives to maximize progress of each worker by tuning parameters that are implicitly exposed by the weakened consistency semantics of the auxiliary data structures. The current implementation tunes three parameters: (a) the replica save period that determines how often elements are saved from native to shadow replica (b) the replica exchange period that determines how often elements are copied from remote workers' shadow replica to native version and (c) sharing set size – the number of elements that are saved and copied between parallel workers. The MemoDyn runtime currently includes an online tuning algorithm based on gradient ascent[3], implemented under an asynchronous model of tuning. In this model, tuning is done concurrently in a separate thread and has minimal interference on the main thread's computation.

Profile-based offline selection of initial parameter values. The initial values of parameters for tuning are determined based on profiling. First, a range of potential initial values for each tunable parameter is selected for offline training. Since the combined space of *all possible* parameter values is prohibitively large, this range is selected by sampling values seen during the *sequential* runs of a set of randomly selected inputs that run for at least one minute. Offline training is then done using a second input set by invoking a *parallelized* version of the search program that performs sparse sharing. In this program, values for each parameter are statically set at the beginning of the program and do not change during execution. This parallel version is invoked for every combination of parameter values computed in the first step, and corresponding speedup is measured. The particular combination of parameter values that results in the best geomean speedup across all inputs is then selected as the initial condition to guide online adaptation. This combination also constitutes the setting for the non-adaptive version of MemoDyn evaluated in Section 6. In our experiments, the profiling step had a geomean overhead of 8.9x over average time for sequential execution across five programs.

Online adaptation using gradient ascent. The online adaptation algorithm used within MemoDyn is based on gradient ascent (Algorithm 1). It is invoked independently for each worker, with the goal of finding the parameter configuration that maximizes the progress made by each worker. The parameters correspond to those exposed by the MemoDyn data structures, and the objective function is a function of these parameters. Starting with the initial values seeded by the offline profiling based parameter selection algorithm, the tuning algorithm first perturbs these values in either direction for each parameter (Lines 4 to 7) and computes the gradient of the objective by measuring the difference in progress made

[3] Other methods like Nelder-Mead [43] did not perform as well in our experiments due to higher overheads.

Algorithm 1: *Online adaptation using gradient ascent*

```
1  def gradient(id, x) :
2      a⃗ ← x⃗
3      for i = 1 to dim(a⃗) do
4          a⃗_i ← x⃗_i + δ_i
5          f1 ← perturbAndMeasure(id, a⃗)
6          a⃗_i ← x⃗_i − δ_i
7          f2 ← perturbAndMeasure(id, a⃗)
8          ▽_i ← (f2 − f1)/(2 × δ_i)
9      end
10 x⃗ ← x⃗_ini  where x⃗_ini is from offline parameter selection
11 for i = 1 to NUMITERS do
12     ▽f ← gradient(id, x⃗)
13     if (‖▽f‖ < ϵ) then
14         return x⃗
15     end
16     x⃗ ← x⃗ + α ∗ ▽f
17 end
18 return x⃗
```

due to this perturbation (Line 8). Because the objective function is not a direct function of the MemoDyn parameters, measurement is performed only after sufficient number of sharing cycles have elapsed beyond the perturbation point (within `perturbAndMeasure` on Lines 5 and 7). Once the gradient has been computed, the parameter configuration is updated in the direction of the gradient, scaled appropriately by a fraction α. This whole cycle is iterated until the gradient norm is very small (which would be the case near a local maximum) or for a fixed number of iterations (Lines 15 to 17).

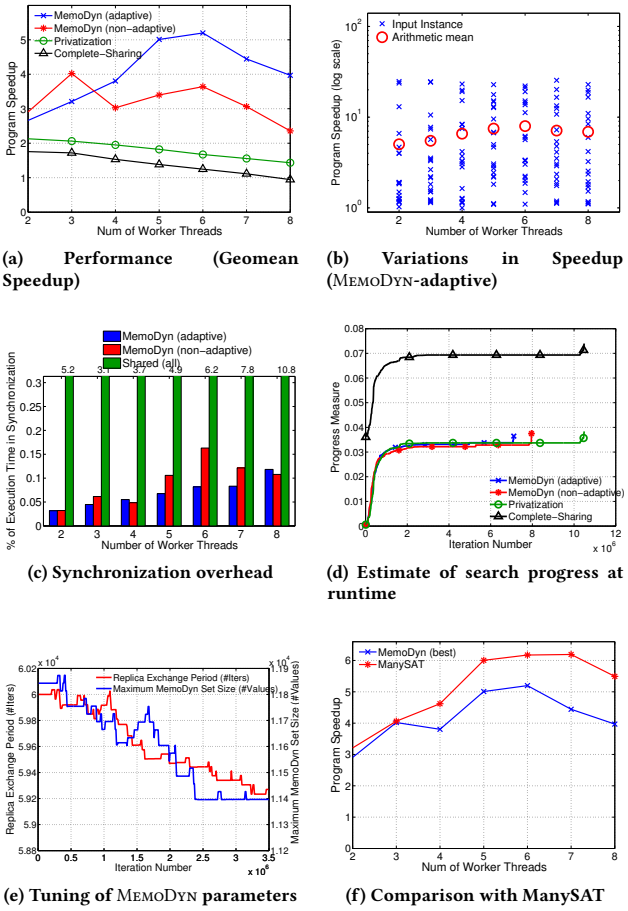
6 Evaluation

MemoDyn is applicable to sequential programs that use auxiliary data structures to memoize results for increased performance, as discussed in Section 1. Thus, MemoDyn is evaluated on five open source sequential search/optimization programs that use memoization, shown in Table 2. These programs were evaluated on multiple randomly selected inputs from well-known open source input repositories that take at least 30 seconds to run. Table 2 also shows the programming effort in number of MemoDyn annotations added and additional changes for implementing standard object oriented abstractions relevant to MemoDyn. These changes only introduce sequential code and use no parallel constructs.

In addition to MemoDyn, two most related non-MemoDyn semantic parallelization schemes were evaluated for comparison: Privatization and Complete-Sharing. Both these schemes have different (non-adaptive) synchronization methods as described in Section 3.1, but are based on the same POSIX-based parallel subsystem as MemoDyn. Figure 4 shows detailed performance results for `minisat`, and Figure 5 shows all other results. The evaluation was done on a 1.6GHz Intel Xeon 64-bit dual-socket quad core machine with 8GB RAM running Linux 2.6.24.

6.1 Boolean satisfiability solver: `minisat`

A total of 12 annotations are inserted to annotate (a) `learnts` as a weakly consistent set, (b) `ClauseAllocator` class as an allocator for elements of the weakly consistent set (c) `progressEstimate` member function, which computes an approximate value for search progress using current search depth and number of solved clauses



(a) Performance (Geomean Speedup)

(b) Variations in Speedup (MemoDyn-adaptive)

(c) Synchronization overhead

(d) Estimate of search progress at runtime

(e) Tuning of MemoDyn parameters

(f) Comparison with ManySAT

Figure 4: MemoDyn **Experimental results for** `minisat`

and assigned variables, as a progress measure. Additionally, a copy constructor was added to `Solver` to enable MemoDyn runtime to perform initial solver replication; an overloaded comparison operator that uses activity heuristics within the solver to rank weakly consistent set elements, and two callbacks to interface garbage collection with the main `Solver` state.

Speedup and Variance. Figure 4a shows the speedup graph for `minisat`. The adaptive version of MemoDyn outperforms the rest by a wide margin. It scales up to six worker threads (a total of twelve POSIX threads, with six additional tuning threads) achieving a geometric mean program speedup of 5.2x over sequential, after which the speedup decreases mainly due to cache interference between multiple parallel workers (14% increase in L2 data cache miss rate from 6 to 7/8 threads). The non-adaptive MemoDyn version achieves better speedup than adaptive MemoDyn for up to three worker threads but slows down beyond that point. Both Privatization and Complete-Sharing versions show poor scaling. In addition to data cache misses, repeated computation of the same learnt clauses across different parallel workers in the former and extremely high synchronization costs in the latter lead to speedup curves with negative slopes.

Program	Description	Total LOC	Source Changes			Profiling Overhead	Input Source	# Inputs	Best Speedup/ Quality	Gain over best	
			# Annot.	Additions/Mods LOC	Additions/Mods Desc					non-adaptive	non-MemoDyn
minisat	SAT Solver	2343	12	36	CC, C, CB	8.4x	Sat-Race	21	5.2x	29.3%	148.0%
ga	Genetic algorithm	811	8	114	CPP, C, P	8.6x	HPEAC	22	3.4x	42.6%	86.1%
qmaxsat	Partial MAX SAT	1783	12	49	CC, C, AO	10.7x	Max-SAT	20	2.6x	24.8%	22.8%
bobcat	Alpha-beta search	5255	8	47	CC, C, P	4.9x	EndGDB	10	3.2x	14.1%	54.2%
ubqp	Quadratic program	1387	8	81	CC, C, P	15x	ACO-Set	15	11%*	1.5%	3.8%

Table 2: Applications evaluated using MemoDyn (CC: Copy constructor, C: Comparison operator, CB: Callback, CPP: C++ conversion, P: Progress Measure, AO: Assignment operator, * Improvement in quality of solution). The final column shows the performance improvement of adaptive-MemoDyn over other best schemes.

Figure 4b shows the variations in speedups across different inputs (in log scale) for the adaptive version. The average of three runs for each input is reported. The speedups range from a minimum of 1.1x to a maximum of 25x; this high variance demonstrates the sensitivity of SAT execution times to input behavior and consequent usefulness of online adaptation in optimizing runtime parallel configuration. Superlinear speedup is due to different total amount of computation needed in the sequential and the parallelized versions. In particular, as noted in section 2, a SAT solver that uses weakly consistent data structures may explore the search space differently from a sequential search. This may result in a more efficient search, more efficient use of memoized results, or even a different satisfying assignment.

Search Progress and Synchronization. Search progress (Figure 4d) improves fastest per iteration for Complete-Sharing, but the corresponding high synchronization costs result in overall slower convergence. The other schemes make relatively slower progress per iteration than Complete-Sharing. The MemoDyn schemes converge the fastest followed by Privatization, as seen from the early termination of their progress curves. Given that Complete-Sharing's synchronization costs (Figure 4c) is an order of magnitude higher than MemoDyn, and Privatization has longer convergence time due to low search space pruning, sparse sharing becomes key to fast convergence times.

Online adaptation. Figure 4e shows a snapshot of adaptation for two MemoDyn parameters at runtime: the replica exchange period and shadow replica set size. The curve for the third parameter, save period, is not shown as it is very similar to exchange period. Initially, the replica exchange period is high, implying a low initial frequency of sharing, but as the search progresses, MemoDyn tuning decreases the value of the replica exchange period. The value for shadow replica size is high in the beginning, but with time the number of elements shared decreases. Overall, tuning in `minisat` causes plenty of elements to be shared less frequently during the initial phases of search; as parallel workers start to converge in later phases, tuning causes *fewer* elements to be shared *more* frequently among parallel workers.

Comparison with Manual Parallelization. ManySAT [20] is a portfolio based, manual parallelization of `minisat`. Figure 4f compares the speedup of the best performing MemoDyn scheme with ManySAT. ManySAT obtains the best speedup of 6.2x over sequential `minisat`. MemoDyn performs competitively, achieving the best speedup of 5.2x. There are several key differences between MemoDyn and ManySAT. First, unlike ManySAT, MemoDyn's multi-search parallel execution model is derived automatically from high-level program semantics without explicit parallelization. Second, ManySAT requires programmer inserted synchronization for implementing clause sharing, whereas MemoDyn's sparse sharing is automatic and is based on an adaptive synchronization protocol. Third, while ManySAT manually throttles clause sharing with a hand-tuned implementation based on feedback control, MemoDyn uses online optimization methods based on application level progress metrics to automatically tune sharing. Thus, MemoDyn's key advantage is its generalized applicability – the use of high level semantic extensions to automate parallelization, synchronization, and tuning gives it the flexibility to change underlying implementations without *any* additional programmer effort.

6.2 Genetic algorithm based Graph Optimization: `ga`

`ga` is a genetic algorithm program originally written in C [21] which we converted to C++. It uses operations like mutation, selection, and crossover to probabilistically create newer generations of candidate solutions based on fitness scores of individuals in the current generation. Solutions are represented as chromosomes within the program. The search terminates on reaching sufficient fitness for a population. The `ga` program in our evaluation additionally uses the concept of elite chromosomes [30], where a non-deterministic fraction of the fittest individuals within certain thresholds are carried forward across generations. MemoDyn annotations were applied to the elite chromosome set. The thresholds on elite chromosomes were automatically enforced by a hard limit defined on the native chromosome set. The correctness of applying MemoDyn annotations follows from the observation that missing a few elite chromosomes can only delay convergence without affecting core algorithm functionality.

However, sharing the best chromosomes between different parallel workers can potentially speed up convergence. Apart from the annotations for weakly consistent sets, code changes involved conversion of C code into C++. The progress measure returns a scaled fitness value of the best chromosome in the current generation. Figure 5a is the speedup graph for `ga`. Adaptive MemoDyn scales up to eight worker threads, achieving a speedup of 3.4x. Although both non-adaptive MemoDyn and Privatization show similar scaling trends, their speedup curves have a much smaller slope than adaptive MemoDyn. Finally, Complete-Sharing shows no speedup

beyond two threads, with synchronization costs at higher thread counts causing a slight slowdown.

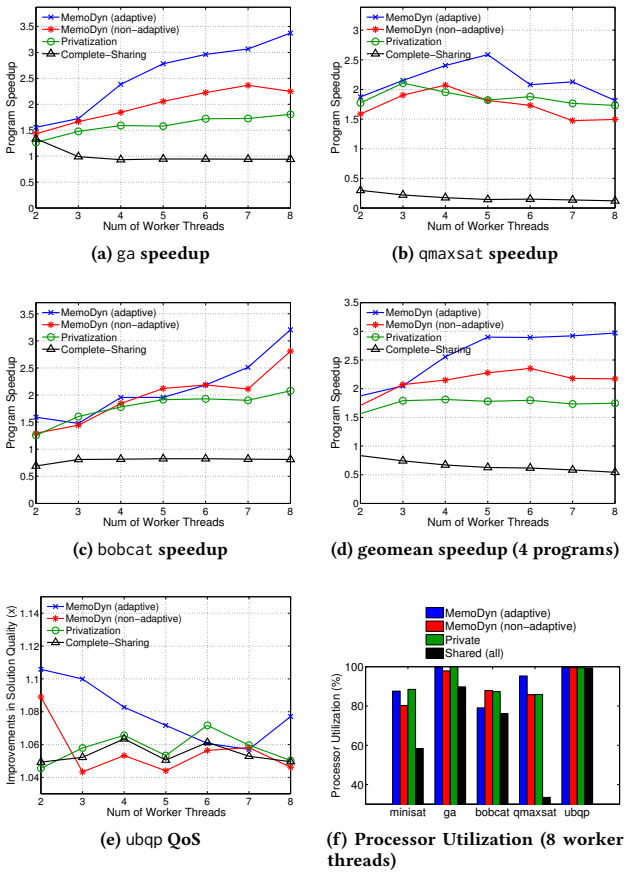
6.3 Partial Maximum Satisfiability Solver: qmaxsat

qmaxsat [26] solves partial MAXSAT, an optimization problem of finding an assignment that *maximizes* the number of satisfied clauses for a given boolean formula. In qmaxsat, SAT is invoked multiple times with different input instances, so learned clauses from newer generations have different variables/clauses compared to previous generations. In context of MemoDyn, this means that when parallel workers progress at different speeds, learned clauses of newer generations belonging to a parallel worker should not be shared and merged with learned clauses of previous generations belonging to other workers. The re-entrant and compositional nature of MemoDyn guarantees correctness transparently by ensuring that parallel SAT invocations across generations always proceed in lock step.

Figure 5b shows the speedup graph. The adaptive version peaks at five threads with a geomean speedup of 2.6x. Interestingly, the Privatization scheme performs better than the non-adaptive MemoDyn scheme although both start to scale down beyond three threads. The synchronization costs for the Complete-Sharing scheme cause a slowdown at all thread counts. Overall, although qmaxsat is long running, learned clause sharing and tuning occur within a shorter window for each of the multiple invocations of SAT within the main program, compared to minisat. The consequent constrained sharing profile together with the cache sensitive nature of parallel SAT are reflected in the performance.

6.4 Alpha-beta search based game engine: bobcat

bobcat [19] is a chess engine based on alpha-beta search. It uses a hash table called the "transposition table" [41] that memoizes the results of a previously computed sequence of moves to prune the search space of a game tree. Using MemoDyn annotations, this table was assigned weakly consistent map semantics. Being a purely auxiliary data structure akin to a cache, partial lookups and weak mutation of the transposition table only cause previously determined positions to be recomputed, without affecting correctness. Apart from adding a copy constructor and a comparison operator that uses the age of a transposition for ranking, the transposition table interface was made generic using C++ templates to expose key and value types. The progress measure employed returns the number of nodes pruned per second weighted by search depth. Figure 5c shows the speedup graphs. In contrast to minisat, the benefits of MemoDyn start only after five worker threads, with the adaptive version achieving the best geomean speedup of 3.2x on eight worker threads compared to the best of 2x for Privatization. Compared to the weakly consistent sets in other programs, the transposition table in bobcat is small in size and is accessed with a high frequency. This causes Complete-Sharing to perform poorly, resulting in a 20% slowdown due its associated high synchronization overhead.



(a) ga **speedup**

(b) qmaxsat **speedup**

(c) bobcat **speedup**

(d) **geomean speedup (4 programs)**

(e) ubqp **QoS**

(f) **Processor Utilization (8 worker threads)**

Figure 5: MemoDyn **Experimental Results II**

6.5 Unconstrained Binary Quadratic Program: ubqp

ubqp [5] solves an unconstrained binary quadratic programming problem using ant colony optimization. The algorithm maintains a population of "ants" that randomly walk the solution space and record history about the fitness of a solution in a "pheromone matrix" structure. Similar to ga, elitism within this program holds a collection of fittest solutions within a set, to which we applied MemoDyn annotations. Instead of a conventional convergence criterion, the search loop in ubqp stops when a given time budget expires. Parallelizing ubqp can improve this program not by reducing its execution time, but by improving the quality of the solution obtained within this fixed time frame. Our evaluation measures the improvement in the quality of the final solution (via an application level metric) of parallel execution over sequential when both are run for a fixed time duration. As seen from Figure 5e, for few threads the adaptive version has better solution quality than the other techniques (11% improvement), but the general trend of all the curves is downward. This is because although MemoDyn enables sharing of fitter solutions among different workers, the pheromone matrix that encodes solutions history is not shared due to high communication costs.

6.6 Discussion

The application of MemoDyn achieves a geomean speedup of 3x on four programs (Figure 5d) and improves the solution quality for one program by up to 11%, while the best non-MemoDyn semantic parallelization scheme (Privatization) obtains a geomean speedup of 1.8x and 7.2% improvement in solution quality, respectively.[4] Regarding programmer effort, an average of 10 MemoDyn annotations and 67 sequential lines of code per program are added or modified to implement C++ abstractions related to MemoDyn. In spite of creating more threads than the number of available hardware contexts, adaptive MemoDyn outperforms other schemes for all evaluated programs. For `ga` and `bobcat`, it attains peak speedup at eight worker threads with sixteen threads in total, outperforming other schemes that create only eight threads in total. Figure 5f shows a much lower processor utilization for Complete-Sharing than other schemes due to synchronization overheads. The utilization for adaptive-MemoDyn is comparable or better than others indicating low parallel execution and context switching overheads.

7 Related Work

Explicit parallelization of search and optimization. Existing parallelizations of search and optimization algorithms are predominantly based on explicit parallelism. SAT solvers have been parallelized for shared memory [20] and clusters [18] using threading and message passing libraries. MemoDyn's semantic sequential language extensions promote easy targeting to multiple parallel substrates without increased programming effort. Additionally, MemoDyn performs online adaptation of parallelized search.

Smart and concurrent data structures. Smart data structures [14] employ online machine learning to optimize throughput of their concurrent operations. STAPL [34] is a parallel version of STL that uses an adaptive runtime. Unlike MemoDyn, these libraries preserve semantics of corresponding sequential data structures. Moreover, MemoDyn's adaptation is based on application level performance metrics (search progress) and aimed at optimizing overall performance and not only to improve data structure throughput.

Data structures with weak semantics. Chakrabarti et al. [8] present distributed data structures with weak semantics for use within a parallel symbolic algebra application. Compared to MemoDyn, these data structures require programmers to explicitly coordinate data transfers, and their semantics enforces eventual reconciliation of mutated data across all parallel workers. WeakHashMap [13] has weak semantics, but unlike MemoDyn it only supports strong deletion and requires manual synchronization and concurrency control. Relaxed data structure synchronization [4, 35, 36, 38] allows races as long as program output is statistically accurate. MemoDyn's sparse sharing provides a form of relaxed synchronization, but is data-race free and adapts to runtime behavior. Cledat et al. [10] leverage the programmer's knowledge about the disjointedness of data access footprints of various computations to parallelize applications. By contrast, MemoDyn explicitly deals with the semantics of access to shared state to enable better parallelization of applications.

Memory consistency models. Various semantics for weakly consistent memory models have been explored at the language [6, 25, 32] and hardware level [1, 2, 12, 16, 40] A memory model depicts the order in which read/write operations to memory locations appear to execute, and addresses the question: "What value can a read of a *memory location* return?". By contrast, MemoDyn is concerned with the order and the semantics of high-level data structure operations, and addresses "What values can a *data structure query* return?". A concurrent implementation of weakly consistent data structures can be achieved on systems implementing a sequential or weak consistency memory model, with correct data structure semantics ensured by appropriate use of low level atomics. In the current MemoDyn implementation, this is realized via the use of `pthreads` locking primitives.

Compiler and runtime support for application adaptation. Adve et al. [3] propose compiler and runtime support for adaptation of distributed applications. In their system, a programmer explicitly parallelizes programs, selects parameters, and inserts calls for tuning at profitable program points. Active Harmony [43] provides an API to specify optimization metrics and to expose parameters that a runtime monitors for online optimization. Rinard et al. [39] present a parallelizing compiler that performs adaptive runtime replication of data objects to minimize synchronization overhead. Parcae [33] is an automatic system for platform-wide dynamic tuning. In comparison, MemoDyn exploits the weak consistency semantics of data structures to optimize parallel runtime configuration by automatically selecting and tuning parameters.

8 Conclusion

This paper presented MemoDyn, a framework for parallelizing search loops with auxiliary data structures that have weak consistency semantics. MemoDyn provides language extensions for expressing weak semantics, and a compiler-runtime system that leverages weak semantics for parallelization and adaptive runtime optimization. Evaluation on eight cores shows that MemoDyn obtains a geomean speedup of 3x on four programs over sequential execution, and an 11% improvement in solution quality of a fifth program having fixed execution time, compared to 1.8x and 7.2% respectively for the best non-MemoDyn semantic parallelization.

9 Acknowledgments

We thank the Liberty Research Group for their support and feedback during this work. We also thank the anonymous reviewers for their insightful comments and suggestions. This work is supported by the National Science Foundation (NSF) through Grants CCF-1814654, CCF-1439085, OCI-1047879, and CNS-0964328. All opinions, findings, conclusions, and recommendations expressed in this paper are those of the Liberty Research Group and do not necessarily reflect the views of the NSF. This work was carried out when the authors were working at Princeton University.

[4] Apart from `minisat`, manual parallelizations for `ga`, `qmaxsat`, `ubqp` either do not exist or are not available online. The manual parallelization of `bobcat` implements Privatization and hence omitted.

References

[1] S.V. Adve and K. Gharachorloo. 1996. Shared Memory Consistency Models: A Tutorial. *Computer* 29, 12 (1996), 66–76.

[2] Sarita V Adve and Hans-J Boehm. 2010. Memory models: a case for rethinking parallel languages and hardware. *Commun. ACM* 53, 8 (2010), 90–101.

[3] Vikram Adve, Vinh Vi Lam, and Brian Ensink. 2001. Language and Compiler Support for Adaptive Distributed Applications. In *Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems (LCTES '01)*. ACM, New York, NY, USA, 238–246. https://doi.org/10.1145/384197.384229

[4] Ankur Agrawal, Jungwook Choi, Kailash Gopalakrishnan, Suyog Gupta, Ravi Nair, Jinwook Oh, Daniel A Prener, Sunil Shukla, Vijayalakshmi Srinivasan, and Zehra Sura. 2016. Approximate computing: Challenges and opportunities. In *Rebooting Computing (ICRC), IEEE International Conference on*. IEEE, 1–8.

[5] C. Blum and M. Dorigo. 2004. The Hyper-Cube Framework for Ant Colony Optimization. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on* 34, 2 (2004), 1161–1172.

[6] Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation (PLDI '08)*. ACM, New York, NY, USA, 68–78. https://doi.org/10.1145/1375581.1375591

[7] Matthew Bridges, Neil Vachharajani, Yun Zhang, Thomas Jablin, and David August. 2007. Revisiting the Sequential Programming Model for Multi-Core. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, Washington, DC, USA, 69–84. https://doi.org/10.1109/MICRO.2007.35

[8] Soumen Chakrabarti and Katherine A. Yelick. 1994. Distributed Data Structures and Algorithms for Gröbner Basis Computation. *Lisp and Symbolic Computation* 7, 2-3 (1994), 147–172.

[9] Romain Cledat, Tushar Kumar, and Santhosh Pande. 2011. Efficiently Speeding up Sequential Computation through N-way Programming Model. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA '11)*. ACM, New York, NY, USA.

[10] Romain Cledat, Kaushik Ravichandran, and Santosh Pande. 2011. Leveraging Data-structure Semantics for Efficient Algorithmic Parallelism. In *Proceedings of the 8th ACM International Conference on Computing Frontiers (CF '11)*. ACM, New York, NY, USA, Article 28, 10 pages. https://doi.org/10.1145/2016604.2016638

[11] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. 2015. Performance Problems You Can Fix: A Dynamic Analysis of Memoization Opportunities. *SIGPLAN Not.* 50, 10 (Oct. 2015), 607–622. https://doi.org/10.1145/2858965.2814290

[12] Joseph Devietti, Jacob Nelson, Tom Bergan, Luis Ceze, and Dan Grossman. 2011. RCDC: a relaxed consistency deterministic computer. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems (ASPLOS '11)*. ACM, New York, NY, USA, 67–78.

[13] Kevin Donnelly, J. J. Hallett, and Assaf Kfoury. 2006. Formal Semantics of Weak References. In *Proceedings of the 5th international symposium on Memory management (ISMM '06)*. ACM, New York, NY, USA, 126–137. https://doi.org/10.1145/1133956.1133974

[14] Jonathan Eastep, David Wingate, and Anant Agarwal. 2011. Smart Data Structures: An Online Machine Learning Approach to Multicore Data Structures. In *Proceedings of the 8th ACM international conference on Autonomic computing (ICAC '11)*. ACM, New York, NY, USA, 11–20. https://doi.org/10.1145/1998582.1998587

[15] Niklas Eén and Niklas Sörensson. 2004. An Extensible SAT-Solver. In *Theory and Applications of Satisfiability Testing*, Enrico Giunchiglia and Armando Tacchella (Eds.). Lecture Notes in Computer Science, Vol. 2919. Springer Berlin / Heidelberg, 333–336.

[16] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 608–621. https://doi.org/10.1145/2837614.2837615

[17] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. 2009. Reducers and Other Cilk++ Hyperobjects. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures (SPAA '09)*. ACM, New York, NY, USA, 79–90. https://doi.org/10.1145/1583991.1584017

[18] Luís Gil, Paulo F. Flores, and Luis Miguel Silveira. 2009. PMSat: A Parallel Version of MiniSAT. *JSAT* 6, 1-3 (2009), 71–98.

[19] Harm Gunnar. 2011. Bobcat. http://github.com/Bobcat/bobcat.

[20] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. 2009. ManySAT: A Parallel SAT Solver. *JSAT* 6, 4 (2009), 245–262.

[21] R. Haney, T. Meuse, J. Kepner, and J. Lebak. 2005. The HPEC challenge benchmark suite. In *HPEC 2005 Workshop*.

[22] E.C.R. Hehner. 1993. *A practical theory of programming*. Springer.

[23] T Joachims. 1999. Making large-Scale SVM Learning Practical. *Advances in Kernel Methods Support Vector Learning* (1999), 169–184. http://www-ai.cs.uni-dortmund.de/DOKUMENTE/joachims_99a.ps.gz

[24] Laxmikant V. Kale and Sanjeev Krishnan. 1993. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications (OOPSLA '93)*. ACM, New York, NY, USA, 91–108.

[25] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-memory Concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 175–189.

[26] M. Koshimura, T. Zhang, H. Fujita, and R. Hasegawa. 2012. QMaxSAT: A Partial Max-SAT Solver system description. *Journal on Satisfiability, Boolean Modeling and Computation* 8 (2012), 95–100.

[27] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. 2007. Optimistic parallelism requires abstractions. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI '07)*. ACM, New York, NY, USA, 211–222.

[28] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, Washington, DC, USA, 75.

[29] Adam Letchford and Andrea Lodi. 2003. An Augment-and-Branch-and-Cut Framework for Mixed 0-1 Programming. In *Combinatorial Optimization - Eureka, You Shrink!*, Michael Jünger, Gerhard Reinelt, and Giovanni Rinaldi (Eds.). Lecture Notes in Computer Science, Vol. 2570. Springer Berlin / Heidelberg, 119–133.

[30] S. Luke. 2010. Essentials of Metaheuristics.

[31] Donald Michie. 1968. Memo Functions and Machine Learning. *Nature* 218, 5136 (1968), 19–22. https://doi.org/10.1038/218019a0

[32] Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. 2016. An Operational Semantics for C/C++11 Concurrency. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 111–128. https://doi.org/10.1145/2983990.2983997

[33] Arun Raman, Ayal Zaks, Jae W. Lee, and David I. August. 2012. Parcae: a system for flexible parallel execution. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 133–144. https://doi.org/10.1145/2254064.2254082

[34] Lawrence Rauchwerger, Francisco Arzu, and Koji Ouchi. 1998. Standard Templates Adaptive Parallel Library (STAPL). In *LCR*. 402–409.

[35] Lakshminarayanan Renganarayana, Vijayalakshmi Srinivasan, Ravi Nair, and Daniel Prener. 2012. Programming with Relaxed Synchronization. In *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES '12)*. ACM, New York, NY, USA, 41–50.

[36] Martin Rinard. 2013. Parallel Synchronization-Free Approximate Data Structure Construction. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Parallelism*. USENIX, San Jose, CA. https://www.usenix.org/conference/hotpar13/workshop-program/presentation/Rinard

[37] Martin C. Rinard. 1994. *The design, implementation and evaluation of Jade, a portable, implicitly parallel programming language*. Ph.D. Dissertation.

[38] Martin C Rinard. 2012. Unsynchronized Techniques for Approximate Parallel Computing. *RACES* (2012).

[39] Martin C. Rinard and Pedro C. Diniz. 2003. Eliminating synchronization bottlenecks using adaptive replication. *ACM Trans. Program. Lang. Syst.* 25, 3 (May 2003), 316–359. https://doi.org/10.1145/641909.641911

[40] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. 2012. Synchronising C/C++ and POWER. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 311–322. https://doi.org/10.1145/2254064.2254102

[41] D.J. Slate. 1987. A chess program that uses transposition table to learn from experience. *International Computer Chess Association Journal* 10, 2 (1987), 59–71.

[42] M. Süß and C. Leopold. 2006. Implementing Irregular Parallel Algorithms with OpenMP. *Euro-Par 2006 Parallel Processing* (2006), 635–644.

[43] V. Tabatabaee, A. Tiwari, and J.K. Hollingsworth. 2005. Parallel Parameter Tuning for Applications with Performance Variability. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*. 57. https://doi.org/10.1109/SC.2005.52

[44] E.G. Talbi. 2006. *Parallel combinatorial optimization*. Vol. 58. Wiley-Blackwell.

[45] George Teodoro and Alan Sussman. 2011. AARTS: low overhead online adaptive auto-tuning. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era (EXADAPT '11)*. ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/2000417.2000418

[46] Abhishek Udupa, Kaushik Rajan, and William Thies. 2011. ALTER: exploiting breakable dependences for parallelization. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation (PLDI '11)*. ACM, New York, NY, USA, 480–491. https://doi.org/10.1145/1993498.1993555

[47] Weixiong Zhang. 2001. Phase Transitions and Backbones of 3-SAT and Maximum 3-SAT. In *Principles and Practice of Constraint Programming - CP 2001*, Toby Walsh (Ed.). Lecture Notes in Computer Science, Vol. 2239. Springer Berlin / Heidelberg, 153–167.