ALTER: Exploiting Breakable Dependences for Parallelization

Abhishek Udupa
University of Pennsylvania
audupa@seas.upenn.edu

Kaushik Rajan
Microsoft Research India
krajan@microsoft.com

William Thies
Microsoft Research India
thies@microsoft.com

Abstract

For decades, compilers have relied on dependence analysis to determine the legality of their transformations. While this conservative approach has enabled many robust optimizations, when it comes to parallelization there are many opportunities that can only be exploited by changing or re-ordering the dependences in the program.

This paper presents ALTER: a system for identifying and enforcing parallelism that violates certain dependences while preserving overall program functionality. Based on programmer annotations, ALTER exploits new parallelism in loops by reordering iterations or allowing stale reads. ALTER can also infer which annotations are likely to benefit the program by using a test-driven framework.

Our evaluation of ALTER demonstrates that it uncovers parallelism that is beyond the reach of existing static and dynamic tools. Across a selection of 12 performance-intensive loops, 9 of which have loop-carried dependences, ALTER obtains an average speedup of 2.0x on 4 cores.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures; D.3.4 [*Programming Languages*]: Processors—Compilers, Optimization, Run-time environments

General Terms Performance, Languages, Experimentation

1. Introduction

Throughout the long history of research on automatic parallelization, much of the progress has been driven by the compiler's ability to accurately detect the dependences between different parts of the program. Such *dependence analysis* has evolved greatly over the years, encompassing representations such as direction vectors, distance vectors, and affine dependences [12] as well as techniques such as array dataflow analysis [26], interprocedural dependence analysis [9] and constraint-based dependence analysis [32]. The precision of dependence analysis has also depended on improvements in alias analysis, shape analysis, and escape analysis, which each have their own rich history.

Despite the tremendous progress in dependence analysis, in practice it remains commonplace for an incomplete understanding of the program's dependences to prohibit seemingly simple transformations, such as parallelization of DOALL loops. There are three fundamental limitations that prevent any dependence analysis from fully solving the problem of automatic parallelization. First, as the general case of dependence analysis is undecidable [36], any static analysis must be conservative in over-approximating the program dependences, thereby under-approximating the opportunities for parallelization. A related problem is that of induction variable analysis: certain dependences can be soundly eliminated via conversion to a closed-form function of the loop index, but complex induction variables (such as iterators through a linked list) are difficult to detect automatically.

Second, even if dependences are precisely identified (e.g., using dynamic dependence analysis [43], program annotations [45], or speculative parallelization [11, 15, 21, 27, 33, 39, 44]), there remain many programs in which memory dependences are accidental artifacts of the implementation and should not inhibit parallelization. For example, it has been shown that in many performance-intensive loops, the only serializing dependences are due to calls to the memory allocator, which can be freely reordered without impacting correctness [7, 41]. Similar dependences are due to benign references to uninitialized data structures, maintenance or output of unordered debugging information, and other patterns [41]. Only by allowing the compiler to reorder or ignore such dependences is it possible to extract parallelism from many programs.

Third, there are many cases in which broken data dependences do change the course of a program's execution, but the algorithm employed is robust to the changes and arrives at an acceptable output anyway. A simple example is the chain of dependences implied by successive calls to a random number generator, which has the potential to serialize many loops. However, in algorithms such as simulated annealing or monte-carlo simulation, any ordering of the calls is permissible so long as each result is random [7, 41]. A more subtle example is that of successive relaxation algorithms, in which the solution is iteratively improved in a monotonic fashion and broken dependences (such as stale reads) may enable parallelism at the expense of a slight increase in convergence time.

In this paper, we make three contributions towards enabling parallelization of programs that are beyond the reach of dependence analysis. First, we propose a new execution model, `StaleReads`, which enables iterations of a loop to execute in parallel by allowing stale reads from a consistent snapshot of the global memory. This model enforces a well-known guarantee called *snapshot isolation* that is frequently employed in the database community as a permissive and high-performance policy for transactional commit. However, while other concepts from databases have impacted programming languages via transactional memory, we are unaware of any exploration or implementation of snapshot isolation as a general-purpose mechanism for loop parallelization. In addition to the basic model, we also offer support for reductions, in which commutative and associative updates to specific variables are merged upon completion of each iteration. We demonstrate that snapshot isolation exposes useful parallelism in several algorithms, in particular for convergence algorithms that are robust to stale reads (as described in the previous paragraph).

PLDI'11, June 4–8, 2011, San Jose, California, USA.

```
while (CheckConvergence(AMatrix, XVector,
                        BVector, dimA) == 0) {
  tripCount++;
  [StaleReads]
  for (int i = 0; i < dimA; i++) {
    sum = 0;
    // scalarProduct reads all of XVector
    sum = scalarProduct(AMatrix[i], XVector);
    sum -= AMatrix[i][i] * XVector[i];
    // write to XVector[i]
    XVector[i] = (BVector[i] - sum) /
                                  AMatrix[i][i];
  }
}
```

Figure 1. Iterative algorithm to solve a system of linear equations of the form Ax=b. The for loop is annotated by ALTER to indicate that the iterations are likely to tolerate stale reads. Parallelization using the ALTER compiler and runtime system gives a speedup of 1.70x on 4 cores (for a sparse matrix A with 40,000 elements).

Our second contribution is a general framework, ALTER, for specifying dependences that do not matter and leveraging that information in a parallel runtime system. Using ALTER, the programmer annotates each loop with one of two permissive execution policies. In addition to `StaleReads`, ALTER supports `OutOfOrder` execution of the loop iterations; this corresponds to the database notion of *conflict serializability* and may break the original dependences so long as execution is consistent with some serial ordering of the iterations. Annotations are also used to declare reduction variables, as described above. Given the annotations, ALTER implements the desired parallelism using a deterministic fork-join model. Each loop iteration (or chunk of iterations) is treated as a transaction, executing in an isolated process and committing with a different policy depending on the model of parallelism. The implementation of ALTER relies on a novel multi-process memory allocator as well as custom collections classes that enable iterators over linked data structures to be recognized as induction variables.

Our third contribution is a practical methodology that leverages ALTER to discover and exploit parallelism. As the ALTER annotations may change the local behavior of a program, it is intractable to infer or verify them using a sound static analysis. However, to assist programmers in exploring the space of potential parallelizations, we propose a test-driven framework that evaluates each possible annotation on each loop in the program and conveys to the programmer the set of annotations that preserve the program output across a test suite. This dynamic analysis, while unsound, can be used to infer likely annotations that are subsequently validated by the programmer. The test-driven framework benefits greatly from ALTER's deterministic execution model, as each test needs to be executed only once. Using this end-to-end methodology, we apply ALTER to assist with parallelization of a set of loops, drawn from Berkeley's parallel dwarfs as well as the STAMP suite. ALTER finds parallelism that is undiscoverable by prior techniques and accelerates performance-intensive loops by an average of 2.0x on four cores.

The rest of this paper is organized as follows. We start with a motivating example for the violation of program dependences during loop parallelization (Section 2). We proceed to describe the ALTER annotation language (Section 3), the implementation of the ALTER compiler and runtime system (Section 4), the annotation inference algorithm (Section 5) and the usage scenarios in which ALTER could be applied (Section 6). We then present our experimental evaluation (Section 7) before wrapping up with related work (Section 8) and conclusions (Section 9).

```
// convergence check simplified for presentation
while (delta > threshold) {
  delta = 0.0;
  [OutOfOrder + Reduction(delta, +)] or
  [StaleReads + Reduction(delta, +)]
  for (i = 0; i < npoints; i++) {
    index = common_findNearestPoint(feature[i],
              nfeatures, clusters, nclusters);
    // If membership changes, increase delta
    // by 1.  membership[i] cannot be
    // changed in other iterations
    if (membership[i] != index) {
      delta += 1.0;
    }
    // Assign the membership to object i
    membership[i] = index;
    // Update new cluster centers :
    // sum of objects located within
    *new_centers_len[index] =
       *new_centers_len[index]) + 1;
    for (j = 0; j < nfeatures; j++) {
      new_centers[index][j] =
         new_centers[index][j] + feature[i][j];
    }
  }
}
```

Figure 2. K-means clustering algorithm from STAMP. ALTER suggests the `OutOfOrder` and `StaleReads` annotations along the for loop, in combination with an additive reduction on the variable *delta* (which is used to determine the termination of the overall algorithm). Parallelizing this algorithm using `StaleReads` gives a speedup of 1.71x on 4 cores.

2. Motivating Examples

This section illustrates two examples where breaking dependences enables parallel execution while preserving the high-level functionality.

The first example (see Figure 1) is a numerical algorithm to solve a system of linear equations of the form $Ax = b$, where A is an $n \times n$ input matrix, b is an input vector of n elements and x is the solution vector of n unknown elements. This algorithm often forms the kernel for solving partial differential equations. The algorithm has two loops, an outer while loop that checks for convergence and an inner for loop that re-calculates each element of *XVector* based on the values of the other elements. The inner loop has a tight dependence chain as the *XVector* element written to in one iteration is read in every subsequent iteration. Thus, the only possible way to parallelize this loop is to violate sequential semantics by not enforcing some of the true dependences.

Based on the results of test cases, ALTER suggests that the inner loop can be parallelized under the `StaleReads` model. While some of the values read from *XVector* will be stale, the algorithm has an outer while loop which checks for convergence and prevents these broken dependences from affecting the overall correctness. This alternate version of the algorithm has been shown in the literature to have the same convergence properties as the sequential version [31]. In fact, this algorithm belongs to a class of algorithms, commonly referred to as algebraic path problems [40], that can tolerate some stale reads. This class includes many popular algorithms such as the Bellman Ford shortest path algorithm, iterative monotonic data-flow analyses, Kleene's algorithm (for deciding whether a regular language is accepted by a finite state automaton) and stencil computations.

Note that it is possible that embracing stale reads can increase the number of (outer loop) iterations needed to converge. For this benchmark, we observe that this increase is quite small in practice.

This is expected as typically the size of *XVector* is large (tens of thousands of elements) while an iteration will read a small number of stale values (at most $(N-1) \times$ *chunkFactor* values on an N-way multicore, where *chunkFactor* is the number of iterations executed per transaction as detailed later). Using ALTER, this parallelization gives a speedup of 1.70x with 4 cores for a problem size of 40,000.

The second example (see Figure 2) shows the main loop of the K-means clustering algorithm. ALTER suggests that the loop can be parallelized with either `OutOfOrder` or `StaleReads` in combination with an additive reduction on the variable *delta*. In this case, `OutOfOrder` and `StaleReads` are equivalent with respect to the program's execution, because every read of a shared variable (*new_centers* and *new_centers_len*) is followed by a write to the same location. Thus, even under the `StaleReads` model there will not be any stale reads in the execution trace, because conflicts in the write sets would cause such concurrent iterations to conflict (this is clarified in the next section). Nonetheless, annotating the loop with `StaleReads` can lead to higher performance than `OutOfOrder`, as it is not necessary to track or compare the read sets within ALTER.

3. Alter Annotation Language

ALTER provides an annotation language for specifying the parallelism in loops (see Figure 3). For all annotated loops, ALTER treats each iteration as a transaction that executes atomically and in isolation. The conditions under which an iteration commits its results are governed by the annotation A, which contains two components: a parallelism policy P and, optionally, a set of reductions R.

The parallelism policy can be one of two types:

1. `OutOfOrder` specifies that ALTER may reorder the loop iterations, so long as the execution is equivalent to some serial ordering of the iterations. This corresponds to the database notion of *conflict serializability*. `OutOfOrder` preserves the semantics of the original program if the iterations are commutative.
2. `StaleReads` is more permissive than `OutOfOrder`: in addition to reordering iterations, the values read in a given iteration may be stale. All stale reads are drawn from a consistent but possibly obsolete snapshot of the global memory. The degree of staleness is bounded: the memory state seen by iteration i, which writes to locations W, is no older than the state committed by the last iteration to write to any location in W. (The only exception is that of reduction variables, which do not count as part of W.)

   This model corresponds to the database notion of *snapshot isolation*; two concurrent iterations can commit so long as their write sets do not overlap. Snapshot isolation is a popular alternative to conflict serializability in the database community, as it leads to better performance. It is adopted by several major database management systems, including InterBase, Oracle, PostgreSQL and Microsoft SQL Server (2005 onward). However, to date there has been no programming language or runtime support for snapshot isolation as a mechanism for loop parallelization.

The second part of an ALTER annotation is an optional set of reductions. Each reduction is specified by the name of a program variable, as well as an operation that is commutative and associative. The annotation asserts that every access to the variable within the loop represents an update using the corresponding operation. For example, if the operation is +, then the variable may appear only in statements of the form *var+=value*. Note that this also prohibits any reads of *var* elsewhere in the loop.

ALTER guarantees that upon completion of the loop, each reduction variable has the value corresponding to a serial execution of the operations that updated it. Note that under the `OutOfOrder` execution model, annotating reductions will not impact the final values assigned to reduction variables, as `OutOfOrder` guarantees that all variables (not just reduction variables) are updated in a manner that is consistent with a serial execution. However, annotating reductions can nonetheless improve the performance under `OutOfOrder`, as iterations that were previously serialized by a dependence on a reduction variable can now execute in parallel. In the case of `StaleReads` execution, annotating reduction variables can impact the final values of those variables (but not any other variables) as well as impacting performance.

$$
\begin{array}{lcl}
A & := & (P, R) \\
P & := & \texttt{OutOfOrder} \mid \texttt{StaleReads} \\
R & := & \epsilon \mid R; R \mid (var, O) \\
O & := & + \mid \times \mid \texttt{max} \mid \texttt{min} \mid \vee \mid \wedge
\end{array}
$$

Figure 3. ALTER annotation language

In addition to the parameters specified in the annotation language, ALTER also allows the user to specify a *chunk factor (cf)*, which dictates the granularity of the execution. While our description above deals in terms of individual iterations, in practice it is beneficial to group multiple iterations together for improving the efficiency of `OutOfOrder` and `StaleReads` execution. The semantics can be understood in terms of chunking: the original loop L is refactored into a nested loop L' in which each iteration of L' represents *cf* iterations of L. The chunk factor can be designated on a per-loop basis, or globally for the entire program.

4. Alter Compiler and Runtime

Given a sequential program and a loop to parallelize, the ALTER compiler produces a concurrent program with embedded calls to the ALTER runtime library. The concurrent program is parameterized by some additional inputs that indicate the semantics to be enforced for each loop. If the user provides annotations for a loop, then these are used to generate the parameters. Otherwise, likely annotations can be inferred by testing their conformance to a test suite (see Section 5). Section 4.2 describes the parameters governing the execution and the assignments needed to exploit the parallelism allowed by the annotations. A salient feature of the ALTER framework is that the output of the compilation process is a deterministic parallel program. Details of how this is achieved are provided in Section 4.3.

4.1 Program Transformation

Program transformations are implemented as phases in the Microsoft Phoenix compiler.

Deterministic Fork Join Model

The ALTER compiler transforms the program to use a process-based fork-join model of computation. The model provides efficient isolation between concurrently executing loop iterations through copy-on-write page protection. Figure 4 shows what the transformed program looks like. The shaded vertices and edges in the figure represent the original sequential control flow.

The high-level functionality of the transformation is to:

1. Create N processes that have identical stack and heap states (apart from ALTER's internal variables), corresponding to the program state on entry to the loop. These N private memories are copy-on-write (COW) mappings of a committed memory state. Thus, just before the loop, there are $N + 1$ versions of memory. In the figure, ❶ and ❷ describe the implementation of this functionality using Win32 system calls.

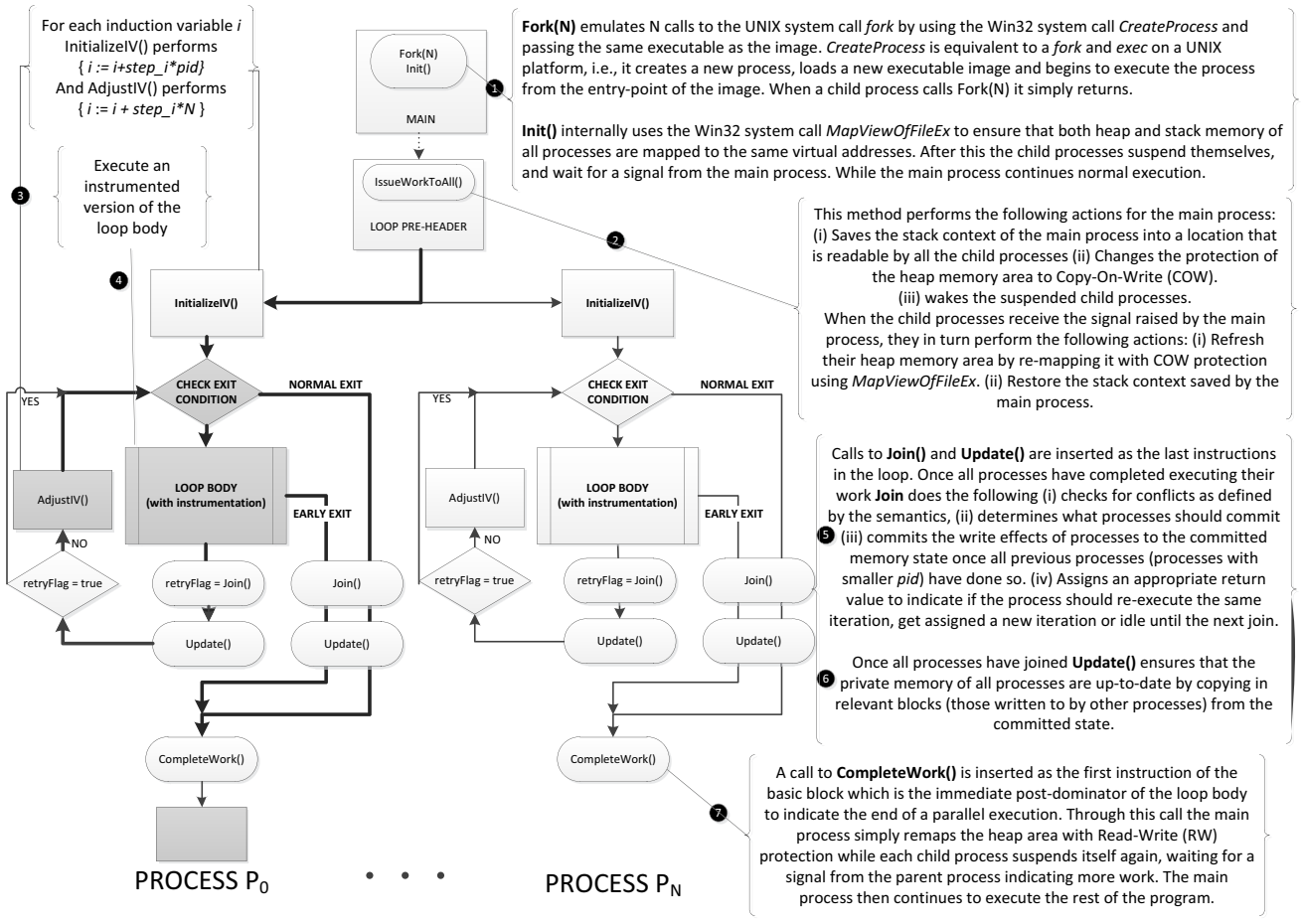

Figure 4. ALTER's fork-join model for executing loop iterations concurrently and in isolation.

2. Distribute iterations to processes and orchestrate a lock-step execution where processes repeatedly: ❸ pickup iterations to execute. ❹ execute them concurrently in isolation on their own private memory while tracking read and write sets (details on instrumentation below). ❺ wait for all processes to complete execution and then depending on validation against execution policy either commit writes to committed memory state or mark the iteration for re-execution next time. Runtime barriers are used to ensure that processes commit changes one after another in deterministic order (ascending order of child pid's). ❻ re-synchronize process private memory with the committed memory state once all processes have committed.

   These steps are repeated until the loop execution is complete.

To keep things simple, Figure 4 depicts the program transformation without chunking. The actual transformation introduces an additional inner loop such that each process executes a consecutive chunk of iterations between joins.

Memory Management

In addition to the above transformation, all calls to memory allocator methods are replaced with calls to the ALTER-allocator. This transformation is essential to ensure that in a multi-process setting, objects can be directly copied between processes without overwriting live values. The ALTER-allocator is designed to be safe and efficient under a multi-process execution environment. It ensures safety by guaranteeing that no two concurrent processes are allocated the same virtual address. We do not describe the design in detail here except to say that it uses techniques similar to ones used by HOARD [5]. Where HOARD is optimized to be efficient in a multi-threaded environment, the ALTER-allocator is designed to be efficient in a multi-process environment. For example, the allocator is optimized to minimally use inter-process semaphores and mutual exclusion primitives. Such inter-process communication primitives are typically much more expensive than locking primitives that can be used in a multi-threaded setting.

ALTER Collection Classes

While simple induction variables can be identified via static analysis, induction variables of loops that iterate over elements of a heap data structure will not be detected by most compilers. To enable parallelization of such loops ALTER exposes a library of standard data structures that are commonly iterated over. When a user wants to parallelize a loop that iterates over such a data structure, she could replace the data-structure with an equivalent ALTER collection class and then use any of the ALTER annotations. ALTER internally manages the state associated with collection classes in a process-safe manner. Note that ALTER collections can also safely be used in a sequential program.

Instrumenting Reads and Writes

The read-write instrumentation performed by the compiler is fairly standard; we briefly describe it here for the sake of completeness. Accesses to both heap memory locations as well as to local

stack variables are instrumented by the compiler by adding calls to runtime methods `InstrumentRead` and `InstrumentWrite`. Heap accesses are instrumented at *allocation* granularity. For instance if a statement of the form `ObjPtr->Field = Value;` is encountered, then the object referenced by `ObjPtr` as a whole is instrumented, assuming that the `ObjPtr` refers to an allocation. The compiler performs the following optimizations to avoid unnecessary instrumentation calls:

- No instrumentation is inserted for a memory access if the object to be instrumented already has a dominating instrumentation.
- If a local variable is defined before its first use, i.e., it is defined afresh in each iteration of the loop, then it is not instrumented, unless it is also *live-out* on loop exit.
- If the loop to be parallelized contains function calls, then accesses to local variables in the function are not instrumented.
- If the memory access to be instrumented matches a pattern of an array indexed by an induction variable, i.e., a loop invariant base pointer indexed by an induction variable, then we instrument the *entire range* once rather than instrumenting each individual array access.

At runtime, the library stores the addresses of the instrumented blocks in a (local) hash set as well as a (global) array. The hash set allows quick elimination of duplicates, while the global array allows other processes to check for conflicts against their respective read- and write-sets.

4.2 Runtime Parameters

As mentioned previously the ALTER compiler translates the sequential program into a concurrent program with some additional configuration parameters. Four different parameters are exposed, combinations of which can be used to enforce various formal guarantees for the loop.

1. The `ConflictPolicy` configuration parameter selects one among four different definitions of conflict, which are applied to all memory locations not corresponding to reduction variables. These four policies, `FULL`, `WAW`, `RAW` and `NONE` form a partial order with respect to the conditions under which they allow processes to commit. These terms have the natural intuitive meaning. `FULL` allows a process to commit only if neither its read nor its write set overlaps with the write set of any of the concurrent processes that committed before it. `WAW` allows a process to commit only if its write set does not overlap with the write set of any of the concurrent processes committed before it. `RAW` allows a process to commit only if its read set does not overlap with the write set of any of the concurrent processes committed before it. `NONE` does not check for any conflicts and allows all processes to commit.
2. The `CommitOrderPolicy` defines whether the iterations of the loop should commit in program order (`InOrder`) or are allowed to commit `OutOfOrder`. Note that even under `OutOfOrder`, iterations commit out of program order only if there are conflicts.
3. `ReductionPolicy` takes a set of $\langle var, op \rangle$ pairs and applies reduction as follows. Let $S_c(x)$ denote the latest value of x in the committed memory. Let $oldS_t(x)$ and $newS_t(x)$ denote the values of x in the private memory of transaction t, at the start and end of its execution. ALTER modifies the state depending on the operation in consideration as follows. (1) if op is idempotent, that is $op \in (\vee, \wedge, \texttt{max}, \texttt{min})$, then the committed memory state is updated as $S_c(x) := S_c(x)\ op\ newS_t(x)$. (2) if $op = +$ then the committed state is updated as $S_c(x) := S_c(x) + newS_t(x) - oldS_t(x)$. $\times$ is handled similarly. Our framework currently only supports these 6 operations. The library also has partial support for programmer-defined reduction operations but this is not fully tested and is not exposed as yet.
4. Finally, `ChunkFactor` takes a integer that defines the chunk factor to be used. We omit this parameter in discussions below and refer to it only when relevant.

Parameters Respecting ALTER Annotations

The following theorems assert that the ALTER annotations, which represent constraints on the parallelism in loops, can be enforced via certain selections of the runtime parameters.

Theorem 4.1. *Executing a loop under* ALTER *with:*

`ConflictPolicy` $=$ `RAW`,
`CommitOrderPolicy` $=$ `OutOfOrder`,
`ReductionPolicy` $=R$

respects the annotation (`OutOfOrder`*, R).*

Proof Sketch The `OutOfOrder` annotation specifies that loop iterations are treated as transactions that commit subject to conflict serializability. ALTER starts concurrent iterations on a consistent memory snapshot and provides isolation for iterations by executing each in its own address space. The `RAW` conflict policy guarantees that a transaction t with location L in its read set will not commit if a concurrent transaction that committed before t has L in its write set, which is the criterion required for conflict serializability. Reordering of iterations is enabled by the `OutOfOrder` ordering policy. □

Theorem 4.2. *Executing a loop under* ALTER *with:*

`ConflictPolicy` $=$ `WAW`,
`CommitOrderPolicy` $=$ `OutOfOrder`,
`ReductionPolicy` $=R$

respects the annotation (`StaleReads`*, R).*

Proof Sketch The `StaleReads` annotation specifies that loop iterations are treated as transactions that commit subject to snapshot isolation. ALTER starts concurrent iterations on a consistent memory state and provides isolation for iterations by executing each in its own address space. The `WAW` conflict policy guarantees that a transaction t with location L in its write set will not commit if a concurrent transaction that committed before t has L in its write set, which is the criterion required for snapshot isolation. □

Parameters Respecting Other Semantics

While our focus in this paper is to explore loop semantics that depart from ordinary execution models, ALTER can also be used to implement well-known models such as safe speculative parallelism and DOALL parallelism. This is asserted by the following theorems:

Theorem 4.3. *Executing a loop under* ALTER *with:*

`ConflictPolicy` $=$ `RAW`,
`CommitOrderPolicy` $=$ `InOrder`,
`ReductionPolicy` $=$ `NONE`

offers safe speculative parallelism (equivalent to sequential semantics).

Proof Sketch The iterations commit `InOrder` while respecting all `RAW` dependences, which implies that none of the dependences in the original program are broken. □

Theorem 4.4. *Executing a loop under* ALTER *with:*

$$\texttt{ConflictPolicy} = *,$$
$$\texttt{CommitOrderPolicy} = *,$$
$$\texttt{ReductionPolicy} = R$$

offers DOALL parallelism with a reduction R.

Proof Sketch As DOALL parallelism applies when there are no dependences between iterations, the `ConflictPolicy` and `CommitOrderPolicy` are not relevant. Nonetheless, reductions can be supported using the ALTER framework. □

While other combinations of the ALTER parameters also lead to sensible execution models, we did not find an analogue for them in practice or application for them in example programs. We leave potential investigation of these models for future work.

4.3 Determinism

A salient feature of ALTER is that given a deterministic sequential program as input ALTER produces a deterministic parallel program. That is, every time the generated executable is run with the same program input and the same values for number of processes N, the chunk factor cf and configuration parameters (`ConflictPolicy`, `CommitOrderPolicy`, `ReductionPolicy`) it produces the same output so long as the program terminates. If the program crashes or does not terminate this also happens deterministically. The runtime avoids races by enforcing runtime barriers.

Determinism follows from the following facts: (1) all concurrent processes have independent memory spaces, (2) no process is allowed to execute `join()` until all processes have completed their work, (3) processes commit their changes to the committed memory state one after another in deterministic order, (4) updates from the committed memory state to private memory spaces occur only after all processes have committed, and (5) the same conflicts are detected for every execution with the same inputs. Determinism is a desirable property for a framework like ALTER that uses a dynamic annotation inference engine with test suite conformance as the validation criterion. Determinism is desirable because the correctness of each test can be determined in a single execution.

5. Annotation Inference

In addition to implementing parallelism consistent with the programmer's annotations, ALTER can also be used to infer a set of annotations that are likely valid for a given loop. Because this inference algorithm is unsound, it is important to use it carefully. We discuss specific application scenarios in the next section.

The annotation inference works via a simple test-driven framework. Given a program, ALTER creates many different versions, each containing a single annotation on a single loop. Together, these versions exhaustively enumerate the ways in which one could add a single annotation to the program (details below). Then, ALTER runs each of these programs on every input in the test suite. Because the ALTER runtime is deterministic, only one execution of each test is needed. Those versions matching the output of the unmodified sequential version (according to a program-specific correctness criterion) are presented to the user as annotations that are likely valid.

To enumerate the candidate annotations, our current implementation works as follows. ALTER systematically explores values for P and R (see Figure 3) by fixing the chunk factor at 16 and evaluating all valid candidates for the other runtime parameters. For each reduction R, a bounded search over reduction variables and operators is performed. The search is restricted to (i) apply one of six reduction operators $\langle +, \times, \texttt{max}, \texttt{min}, \wedge, \vee \rangle$, and (ii) apply the same reduction operator (or lack thereof) to all stack variables[1]. Also a search for a valid reduction is performed only if none of the annotations of the form (P, ϵ) are valid.

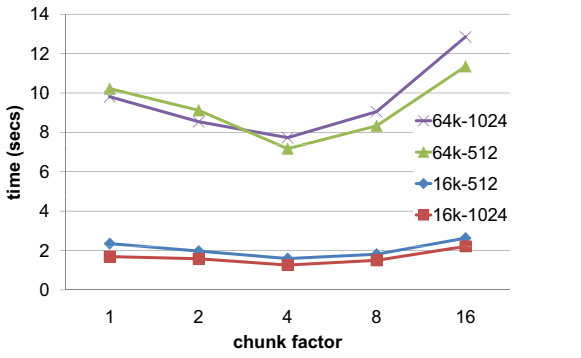

Figure 5. Performance impact of the chunk factor, for various input sizes (numbers of points and clusters) on the K-means benchmark.

After ascertaining valid annotations, an iterative doubling algorithm is used to the find an appropriate chunk factor. Starting from a candidate value of 1 the chunk factor is iteratively doubled until a performance degradation is seen over two successive increments. The candidate that led to the best performance is then chosen as the chunk factor. Our results indicate that the chunk factor is dependent only on the loop structure and not on the inputs. Figure 5 shows how performance varies with increasing chunk factors for different inputs of *K-means*. As can be seen the best performing chunk factor for all 4 inputs remains the same. Similar behavior is observed for other benchmarks.

In addition to reporting the set of annotations that lead to valid executions, ALTER also gives hints as to the cause of failure for annotations that are unsuccessful. For each annotation, the reported outcome is one of the following: *success, failure* $\in$ *(crash, timeout, high conflicts, output mismatch)*. *Success* represents the case where the execution produces a valid output. Failure cases can be classified into cases where no output is produced (*crash, timeout*) and cases where an output is produced but it is either incorrect (*output mismatch*) or is unlikely to lead to performance improvements (*high conflicts*). A *timeout* is flagged if the execution takes more than 10 times the sequential execution time. An execution is flagged as having high conflicts if more than 50% of the attempted commits fail. Both of these behaviors are correlated with performance degradation and hence we deem them as failures.

6. Usage Scenarios

To explore concrete applications of ALTER in real-world contexts, we consider three scenarios – manual parallelization, assisted parallelization, and automatic parallelization. While our primary target is assisted parallelization, under certain circumstances ALTER can also make sense for purely manual or automatic parallelization. We summarize these scenarios in Table 1 and expand on them below.

Assisted parallelization In this scenario, a user is attempting to parallelize a code base for which she has partial but incomplete knowledge. For example, perhaps the user has authored part of the system, but relies on third-party libraries as well. In such a situation, ALTER serves a natural role in assisting the user to parallelize

[1] This is sufficient for our current benchmarks. It should be fairly straightforward to extend the search strategy to explore different reductions for different variables. Because reduction variables are not allowed to interact, either with each other or with the program variables, each variable can be tested independently without leading to a combinatorial explosion.

	Manual Parallelization	Assisted Parallelization	Automatic Parallelization
User has:	deep understanding of code	some familiarity with code	little or no familiarity with code
Annotations are:	written by hand	inferred but checked by hand	inferred and used as-is
ALTER **serves as:**	1) high-level parallelism library 2) tool for rapid prototyping	1) set of hints to investigate further 2) upper bound for discoverable parallelism	1) temporary substitute for human parallelizer 2) unsound parallelizer for tolerant environments

Table 1. Usage scenarios for ALTER. While ALTER is intended primarily for assisted parallelization, it could also find application in purely manual or automatic parallelization.

the code. By evaluating various loop annotations via executions on test suites, ALTER can suggest which models of parallelism are likely to be valid and beneficial for each loop in the program. The user can investigate these suggestions to determine whether or not they are sound before integrating them into source base. In addition, ALTER's suggestions could serve as a useful upper bound for certain kinds of parallelism in the program. If ALTER does not find likely parallelism in any loop, then perhaps a new algorithm will be needed to leverage multicore hardware.

Manual parallelization This scenario applies to cases in which the user of ALTER is authoring a new parallel program or is already equipped with a deep understanding of an existing code base. In this case, the user can think of ALTER as providing an API for exploiting known parallelism in the program. All of the annotations are written by the user herself. The ALTER runtime system could be shipped as part of a product, or applied only for internal prototyping in order to quickly explore the speedups offered by various transformations. In the latter case, the most beneficial transformations may be re-written manually (without the ALTER runtime system) to further customize and optimize the performance.

Automatic parallelization In the final scenario (which remains largely hypothetical), ALTER is applied as an autonomous parallelization engine. Perhaps this could make sense for obscure legacy codes, in which it is unduly expensive for new developers to fully understand the intricacies of the implementation. ALTER could be used to infer parallel annotations by evaluating them for correctness across a large and rigorous test suite. In the case of legacy codes, test suites often provide the only arbiter as to whether changes in the environment have violated implicit assumptions of the code. Just as many human intuitions about very complex systems can only be verified via testing, there might be cases where it is not unreasonable to utilize testing as the sole correctness criterion for a compiler-directed transformation.

Nonetheless, to make sense in a production environment, additional flexibility is likely needed to embrace ALTER as an automatic parallelization engine. For example, if parallel speedups are urgently needed to proceed with whole-system prototyping (e.g., to evaluate the user experience of a new gaming engine), then perhaps ALTER could be applied temporarily until a human developer has a chance to step in and laboriously verify the changes. Alternately, there may be end-user applications where speed is more important than correctness. For example, one may desire a low-latency preview of multimedia content before the official decoder has finished rendering it, even if the preview has a chance of being wrong. Or, on a mobile platform, if the battery is low one may want to quickly execute some application (with some chance of failure) rather than initiating a slower execution that is guaranteed to timeout due to a dead battery.

7. Experimental Evaluation

We evaluate ALTER with algorithms from Berkeley's dwarfs [4] and sequential versions of the STAMP [28] benchmarks (see Table 2). A dwarf represents an algorithmic method that captures a commonly used pattern of computation. We utilize all dwarfs for which we could find a good representative algorithm and all STAMP benchmarks that we could get to compile on Windows[2].

Table 2 describes the benchmarks used. Our transformations target a single loop in each benchmark. For many benchmarks the main computation happens in a single loop nest; we report results for the nesting level that leads to the most parallelism. For benchmarks containing multiple important loops, the targeted loop is indicated in the *description* column of the table. For each benchmark, we use test inputs to infer the annotations and then use a larger input (shown in bold) to measure performance. All performance measurements are conducted on an 8-core Intel Xeon machine ($2 \times quad$ core at 2.4GHz) with a 64-bit Windows Server 2008 operating system.

7.1 Results of Annotation Inference

We utilized the annotation inference algorithm to infer all annotations used in our evaluation. Correctness of the program output was evaluated on a case-by-case basis. For 4 of the benchmarks (*Labyrinth*, *Genome*, *GSdense*, *GSsparse*) assertions in the code validate the output. For the remaining programs, we utilized a program-specific output validation script, which often made approximate comparisons between floating-point values.

The results from the inference algorithm are reported in Table 3. In addition to the models exposed by our language we also check if the program can be parallelized by some existing auto-parallelization techniques. We add a check in `join()` to see if the loop has any loop-carried dependences. The results of this check are shown in column *dep*. We also check to see if the loop is amenable to thread level speculation (`TLS`). The outcome of this check could be one among (*success, failure* $\in$ *(crash, timeout, high conflicts)*).

Interestingly, we find that in all cases a single test is sufficient to identify incorrect annotations. We find that when `TLS`, `OutOfOrder` or `StaleReads` fail, they fail either due to timeouts or due to a large number of conflicts. The only exception is *AggloClust*, where the application crashes under `OutOfOrder` and `TLS`. In these two cases the machine runs out of memory (due to very large read sets). We find that an incorrect reduction leads either to an invalid output or a timeout. An interesting case is the $+$ reduction for *SG3D*. The convergence check in this algorithm looks for $max_{\forall i}(error_i) < threshold$ so a `max` reduction works. A $+$ leads to a check of the form $\sum_{\forall i}(error_i) < threshold$, which also produces a valid output but convergence takes much longer. Other reductions lead to a deviation of $\pm 0.01\%$ from sequential output in some entries.

Overall we find that all but one benchmark (*Labyrinth*) can be parallelized with ALTER. Both *SSCA2* and *Genome*, which are known to be amenable to `OutOfOrder` [28], also lead to a correct execution under `StaleReads`. This is because all variables that are read in the loop are also written to. Hence it is sufficient to check for WAW conflicts alone and no read instrumentation is required. Though it is known that *K-means* can be parallelized with `OutOfOrder` we find that it leads to a very slow execution. We find that the only practical execution model for *K-means* is to use a combination of `StaleReads` and $+$ reduction.

[2] Compiling STAMP benchmarks on Windows requires a substantial manual effort. We succeeded in compiling 4 benchmarks out of the 8 benchmarks in the suite; we present full results for these 4 cases.

BENCHMARK	DESCRIPTION	LOOP WGT	INPUT SIZE (inference; **benchmarking**)
Genome (STAMP)	The genome sequencing algorithm from STAMP is described in detail in [28]. We parallelize the first step (remove duplicate sequences).	89%	4M segments; **16M segments**
SSCA2 (STAMP)	This benchmark includes a set of graph kernels (details in STAMP [28]). We focus on the second loop in kernel 1.	76%*	18; 19; **20 (problem scale)**
K-means (STAMP)	The K-means algorithm is a popular clustering algorithm (we use the implementation from STAMP [28]). The main loop in the algorithm recomputes the association of points to clusters until convergence (see Figure 2).	89%	16K pts, 512 clusts; 16K pts, 1024 clusts; **64K pts, 512 clusts; 64K pts, 1024 clusts**
Labyrinth (STAMP) *uses* ALTER*Vector*	This algorithm does an efficient routing through a mesh (details in STAMP [28]). An ALTERVector is used here.	99%	$128 \times 128 \times 3$ grid, 128 paths; $\mathbf{256 \times 256 \times 5}$ **grid, 256 paths**
AggloClust (Branch and bound) *uses* ALTER*List*	The agglomerative clustering algorithm utilizes a special tree (k-d tree) to efficiently bound *nearest neighbor* searches in a multi-dimensional space. Our implementation is adapted (C++ instead of Java) from Lonestar [22]. We simplify the original implementation by not updating the bounding boxes on k-d tree node deletions. We focus on the main loop of the program ([22] has the pseudo-code). As the loop iterates over a list we replace the sequential list with an ALTERList.	89%	100K pts; **1M pts**
GSdense (Dense linear algebra)	We use the Gauss-Seidel iterative method [31] for solving a system of equations (refer Figure 1). Depending on whether the A matrix is sparse or dense it uses two different representations of the matrix. As noted before, violating some true dependences still preserves the overall functionality and provides the same convergence guarantees.	100%	10000×10000; $\mathbf{20000 \times 20000}$
GSsparse (Sparse linear algebra)		100%	20000×20000; $\mathbf{40000 \times 40000}$
Floyd (Dynamic programming)	The Floyd-Warshall algorithm uses a triply nested loop (on k, i, and j) within which it repeatedly applies the relaxation $path[i][j] := min(path[i][j], path[i][k] + path[k][j])$. Though the loop has a tight dependence chain, it turns out that even if some true dependences are violated, all possible paths between each pair of vertices are still evaluated [40].	100%	1000 nodes; **2000 nodes**
SG3D (Structured grids)	The algorithm uses a 27-point three-dimensional stencil computation to solve a partial differential equation [13]. A triply-nested inner loop iterates over points in 3D space, updating their value and tracking the maximum change (*error*) that occurs at any point. An outer loop tests for convergence by checking if the *error* is less than a given threshold. While the stencil computations can tolerate stale reads, the update of the *error* value must not violate any dependences, or the execution could terminate incorrectly.	96%	$64 \times 64 \times 64$; $\mathbf{128 \times 128 \times 128}$
BarnesHut (N-body methods) *uses* ALTER*List*	The Barnes-Hut algorithm uses a quad-tree data structure to solve the N-body problem. We use the implementation from Olden [10]. We parallelize the main loop that iterates over a list by transforming it to use an ALTERList.	99.6%	4096 particles; **8192 particles**
FFT (Spectral methods)	We utilize the two-dimensional iterative FFT solver from [1].	100%*	1024, **2048**
HMM (Graphical models)	We use the Hidden Markov Model solver from [1].	100%	512, **1024**

* *FFT has two identical loops each taking 50% of the execution time. The annotations inferred and speedups obtained apply to both these loops. SSCA2 has an initial random input generation step. We do not include the time taken for random input generation.*

Table 2. Benchmarks used for evaluation. Eight benchmarks are drawn from Berkeley's dwarfs; the algorithm name is subtitled with the dwarf that it represents. Four benchmarks are drawn from STAMP. The "LOOP WGT" (loop weight) column indicates the fraction of the program's runtime that is spent in the loop targeted by ALTER.

Benchmark	Dep	TLS	OutOrd	Stale	Reduction
Genome	Yes	success	success	success	N/A
SSCA2	Yes	timeout	success	success	N/A
K-means	Yes	h.c.	h.c.	h.c.	+
Labyrinth	Yes	h.c.	h.c.	h.c.	N/A
AggloClust	Yes	crash	crash	success	N/A
GSdense	Yes	timeout	timeout	success	N/A
GSsparse	Yes	timeout	timeout	success	N/A
Floyd	Yes	timeout	timeout	success	N/A
SG3D	Yes	h.c.	h.c.	h.c.	max/+
BarnesHut	No	success	success	success	N/A
FFT	No	success	success	success	N/A
HMM	No	success	success	success	N/A

Table 3. Results of annotation inference. The table shows whether there is a dependence carried by the loop (**Dep**) as well as the results for thread-level speculation (**TLS**), `OutOfOrder` execution (**OutOrd**), and `StaleReads` (**Stale**). High conflict results are abbreviated as h.c. Reductions are shown where applicable.

Benchmark	cf	Transaction Count	RW Set / Trans.	Retry Rate
Genome-StaleReads	4096	32768	16	0.2%
Genome-OutOfOrder	4096	32768	89	0.2%
Genome-TLS	4096	32768	90	0.16%
SSCA2-StaleReads	64	16384	277	3.5%
SSCA2-OutOfOrder	64	16384	6340	3.5%
K-means-1024	4	65552	136	3.4%
K-means-512	4	81940	136	6.3%
AggloClust	64	46608	28	3.6%
GSdense	32	2720	62	0
GSsparse	32	12580	32	0
Floyd	256	24576	428	0
SG3D	4	23560	208	0

Table 4. Instrumentation details for representative benchmarks. The columns show the chunk factor (**cf**), the number of chunks (transactions) executed (**Transaction Count**), the average size of the read and write set, in words, per transaction (**RW Set / Trans.**), and the average rate at which iterations fail to commit (**Retry rate**).

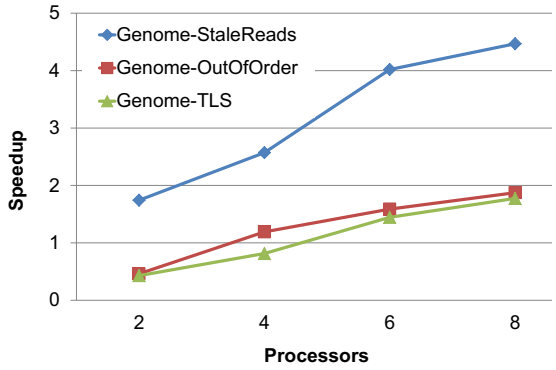

Figure 6. Genome.

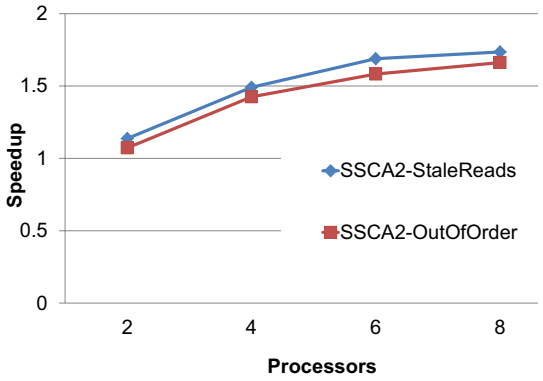

Figure 7. SSCA2.

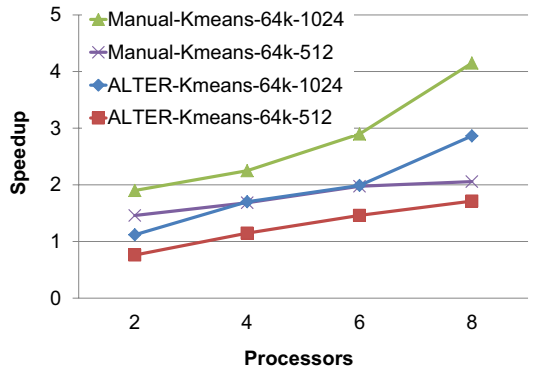

Figure 8. K-means. The plot compares results with manual parallelization on two different inputs.

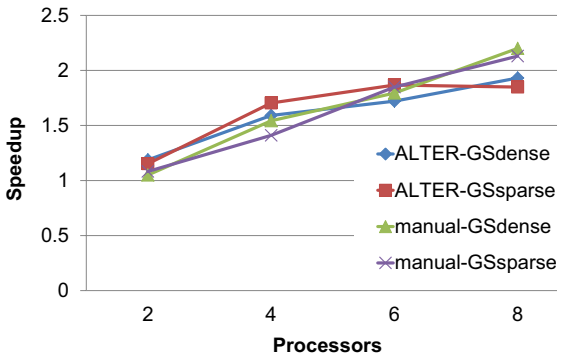

Figure 9. Dense and sparse linear algebra. The plot compares results with manual parallelization for both dense and sparse cases.

Four other benchmarks (*GSdense*, *GSsparse*, *Floyd* and *SG3D*) are tolerant to stale reads. Among these, SSG3D needs a combination of `StaleReads` and reduction for parallelization while the other three do not need reduction. Only three benchmarks have no loop carried dependences and (of those with dependences) only one is amenable to efficient speculation.

7.2 Performance Results

The results for all annotations that lead to a valid execution during testing are shown in Figures 6 through 13. Each figure shows the speedup over the original sequential execution (without ALTER) for the loop nest being parallelized.

A maximum speedup of up to 4.5X is observed (ignoring "trivial" benchmarks with no loop carried dependences) with 8 cores. For the two benchmarks (*Genome* and *SSCA2*) that are amenable to multiple annotations, we find that `StaleReads` leads to much better performance than `OutOfOrder`. This is because enforcing `StaleReads` does not need read instrumentation. We report the size of the read and write sets per transaction as well as retry rates in Table 4. As can be seen by comparing the amount of instrumentation for `OutOfOrder` and `StaleReads` for a given benchmark, there is a much larger number of reads (instrumented by `OutOfOrder`) than writes (instrumented by both `OutOfOrder` and `StaleReads`) within a transaction[3]. Further, we find that for *Genome* TLS performs nearly as well as `OutOfOrder` but not as well as `StaleReads`.

We find that the speedup obtained for *K-means* depends on the number of clusters to be formed. The larger the number of clusters to be formed, the fewer the conflicts. As can be seen from Figure 8, when the number of clusters decreases from 1024 to 512 the speedup comes down from 2.8X to 1.7X. So long as the probability that two points map to the same cluster is low, the algorithm should see a speedup. Further note that while the speedup varies, the best parallelism policy or chunk factor does not change with input. As seen before, both inference and evaluation inputs perform best at the same chunk factor.

Benchmarks *GSdense*, *GSsparse*, and *Floyd* when executed under `StaleReads` lead to no conflicts. This is because while these loops have many RAW dependences there are no loop-carried WAW dependences. We also find that breaking RAW dependences hardly increases the number of iterations needed to converge (GSdense increases from 16 to 17, while GSsparse increases from 20 to 21). Unfortunately, both *GSdense* and *GSsparse* are memory bound and hence do not scale well beyond 4 cores. Like the above bench-

[3] Though STAMP benchmarks come along with instrumentation hints, we ignore them and use our own static analysis to embed instrumentation in the program. Because this analysis is automatic and does not depend on any special knowledge of the program, the overheads observed are likely to generalize to other programs.

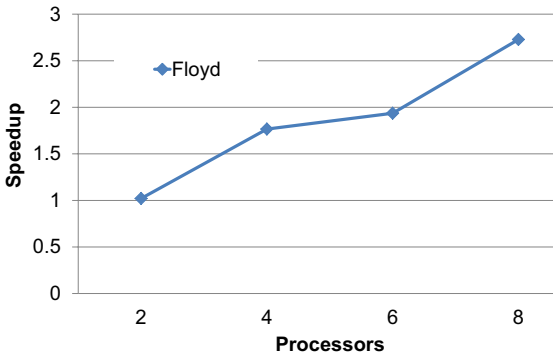

Figure 10. Floyd Warshall algorithm.

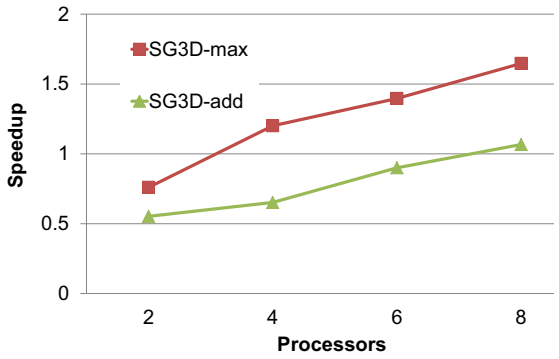

Figure 11. SG3D: 27 point 3D stencil, with alternate reductions.

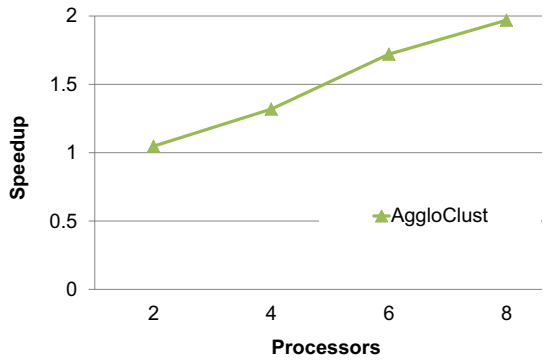

Figure 12. Agglomerative Clustering.

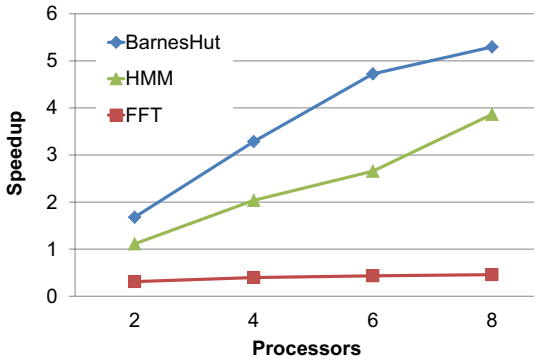

Figure 13. BarnesHut, FFT, and HMM.

marks *SG3D* leads to no conflicts under `StaleReads` with reduction. Among `max` and + reduction we find that using + degrades performance as it leads to a significant increase in the number of iterations to converge (1670 to 2752).

Among the three benchmarks that have no loop carried dependences, ALTER provides reasonable speedups for *HMM* and *barnesHut* but slows down *FFT*. The slowdown on FFT is due to high instrumentation overhead. FFT uses a `complex` data type, which results in many copy constructors that are instrumented by ALTER. This effect could be avoided by a more precise alias analysis or via conversion of complex types to primitive types.

7.3 Comparison with Manual Parallelization

Finally, we manually parallelize two of the benchmarks in order to provide a realistic parallel baseline for ALTER. We manually implement a multi-threaded version of Gauss-Seidel that mimics the runtime behavior of `StaleReads` by maintaining multiple copies of *XVector* that are synchronized in exactly the same way as a chunked execution under ALTER. We also parallelize *K-means* using threads and fine-grained locking.

As shown in Figure 9, ALTER performs comparably to manual parallelization on Gauss-Seidel. However, on *K-means*, it performs 20-47% slower than manual parallelization (considering all tests between 4 and 8 cores) as shown in Figure 8. This slowdown is due to the overhead of the ALTER runtime system as it explores parallelism via optimistic, coarse-grained execution rather than pessimistic fine-grained locking.

8. Related Work

Parallelizing compilers Compilers rely on static analysis to identify parts of the program that can be parallelized. Often they target simple forms of loop parallelism like DOALL and pipeline parallel loops or loops with simple reductions [17]. Parallelizing compilers are typically restricted by the accuracy of static analyses such as data dependence analysis and alias analysis [20]. Richer forms of parallelism can be identified via commutativity analysis [2, 38] that detects if two program tasks commute with each other or not. The analysis needs to be able to prove the equivalence of the final memory state with either order of execution of the tasks before applying a parallelizing transformation. One of the contributions of ALTER is to provide a test-driven framework to identify whether iterations of a loop are likely to commute with each other.

The classic compiler transformation of *privatization* is similar to `StaleReads` in that it enables loop iterations to operate on local copies of global variables [26, 34]. However, there is a key difference between privatization and `StaleReads`: privatization does not allow loop iterations to communicate via the privatized variables. The key aspect of `StaleReads` is that loop iterations *do* communicate, but the values read might be stale, reflecting the update of an earlier iteration rather than the most recent one.

Implicitly parallel programming Rather than providing explicit directives as to the allowable parallelism, as in OpenMP or ALTER, an alternative approach is to utilize an implicitly parallel programming model that enables the compiler to automatically identify and exploit the parallelism [19]. Of the many efforts

in this space, some have also proposed an annotation language to prevent spurious dependences from inhibiting parallelization. Bridges et al. advocate using annotations for commutative functions and non-deterministic branches to enable parallelization of legacy code [7]; their "commutative" annotation plays a similar role to our `OutOfOrder` annotation in allowing certain dependences to be reordered. In the Paralax infrastructure [45], the programmer annotates functions, function arguments, and variables with extra information regarding dependences; for example, a "kill" annotation indicates that a given variable is no longer live. The system can infer likely annotations using a dynamic analysis. However, there is a key difference between this inference algorithm and ours: the Paralax system aims to infer annotations that precisely describe the sequential execution, but could not be identified without testing. In contrast, our testing procedure identifies opportunities to *violate* the sequential semantics while maintaining end-to-end functionality.

Parallel runtime systems Systems such as OpenMP allow the programmer to indicate parallelism via a set of annotations, in a manner analogous to ALTER. However, the parallelism offered by ALTER can not be implemented using OpenMP directives. While OpenMP supports DOALL loops and privatized variables, ALTER provides isolated and potentially stale reads of state that is communicated between loop iterations. ALTER is also very different from OpenMP in that (i) ALTER supports various notions of conflicts and provides runtime support to roll back and retry iterations, (ii) it efficiently and automatically handles full state replication without requiring annotations for shared and private variables, and (iii) ALTER is deterministic: for a given thread count and chunk factor, ALTER always produces the same output for a given input, while parallelization with OpenMP may admit non-deterministic data races.

STAPL [3] is a runtime library that provides parallel algorithms and containers for collection classes; it is a superset of the standard template library (STL) in C++. STAPL overlaps our goal of providing customized collection classes that are suitable for parallelization. However, this is only a small piece of our overall system. We utilize the custom collection classes to enable a new execution model and a deterministic parallel runtime for discovering hidden parallelism via testing.

We note that there are other parallelization frameworks that use a process-based model to ensure isolation. These include (1) the behavior oriented parallelization framework [15] that uses process-based isolation for speculative parallelization, and (2) the Grace runtime system [6] that forks off threads as processes to avoid common atomicity related concurrency errors.

Test-driven parallelization Closely related to our work is the QuickStep system for parallelizing programs with statistical accuracy tests [29]. QuickStep searches for DOALL parallelism by annotating loops with OpenMP directives and evaluating if the resulting parallelization is acceptably accurate for the inputs in the test suite. Quickstep also explores synchronization, replication, and caching transformations that could improve accuracy or performance, and provides metrics and tools to help programmers find an acceptable parallelization. Both systems share the philosophy of violating dependences to enable new parallelism detection. One difference is in the run-time system: while QuickStep utilizes OpenMP, we propose a new runtime system that enables the `StaleReads` execution model and other benefits (see above). A second difference is that ALTER is designed to provide deterministic execution and freedom from data races, while QuickStep is designed to generate potentially nondeterministic parallel programs that may contain data races. As a consequence, ALTER uses a single execution to verify correctness on a given input, while QuickStep performs multiple test executions on the same input to determine if a candidate parallelization respects the accuracy bounds with acceptable probability.

Profile-driven and speculative parallelization Profile driven parallelization tools augment compiler driven transformations by using profile inputs to identify `DoAll` and pipeline parallelism in programs [30, 41, 43]. Speculative parallelization [11, 15, 27, 33, 35, 44] is a related form of parallelization that can be enabled by profile-driven tools. Execution of speculatively parallelized programs typically requires support for thread level speculation either in hardware [21, 39] or in software [11, 15, 27, 35, 42]. To the best of our knowledge, all existing speculative and profile-driven tools that exploit inter-iteration parallelism in loops are restricted to guarantee sequential semantics for the loop. By contrast, ALTER exposes and exploits alternative semantics, including the ability to permit (bounded) stale reads for certain values while nonetheless preserving overall functionality. In this way, the `StaleReads` execution model is fundamentally different from speculation. While frameworks such as behavior oriented parallelism [15] may have similar goals at the abstract level, thus far (in their concrete form) they have utilized speculation that respects program dependences.

Transactional memory systems Software transactions have been proposed as an alternative to traditional lock-based programming. A transactional memory system [18, 25] is needed to support concurrent execution of software transactions. Transactional memory systems try to provide conflict serializability as the correctness criterion, however the exact semantics provided by most STMs is captured better by a correctness guarantee referred to as `opacity` [16]. Recently, researchers have also explored snapshot isolation as a correctness criterion in STMs [14, 37]; however, the experiments to date have utilized a transactional programming model where the programmer thinks in terms of explicit transactions. ALTER targets existing software and helps to identify whether loops can tolerate stale reads and reordering of iterations.

Language constructs for expressing parallelism Many researchers have investigated new programming language constructs for expressing parallelism. Most of these constructs introduce richer semantics that cannot be detected by existing parallelization tools. The Galois programming system introduces constructs to iterate over sets (optimistic iterators) and allows programmers to specify conditions under which methods commute [23, 24]. With these constructs a programmer can write loops whose iterations can execute optimistically in parallel and commit out of order as long as there are no commutativity violations. The revisions programming model [8] exposes language constructs through which a programmer can assign special isolation types to variables and specify merge functions for them. These merge functions are applied at the end of a parallel section of computation. ALTER's contribution is in determining if existing sequential programs are amenable to richer forms of parallelism, without intervention on the part of the programmer. It also proposes the `StaleReads` execution model, which to our knowledge has not been used as a general approach for loop parallelization.

9. Conclusions

Despite decades of research on automatic parallelization, the results achievable by a compiler are rarely comparable to that of a human. We believe that one of the primary limitations holding compilers back is their conservative treatment of memory dependences. Though humans can and do re-arrange dependences – which are often artifacts of the implementation, or non-essential elements of the algorithm – compilers, for the most part, do not.

In this paper, we take a first step towards bridging this gap by proposing ALTER: a system that violates certain memory dependences in order to expose and exploit parallelism in loops. We em-

phasize that ALTER is not intended to completely replace a human; as its inferences are unsound, we still rely on human judgement to drive the parallelization process. However, ALTER could greatly assist the developer by providing a new set of parallelism primitives as well as suggestions regarding their most effective use.

Our evaluation indicates that the parallelism exploited by ALTER is beyond the reach of existing static and dynamic tools. In particular, for one third of our benchmarks, the use of snapshot isolation – allowing stale reads within loops – enables parallel execution even when prior techniques such as speculative parallelism and out-of-order commit do not.

Acknowledgments

We are very grateful to Sriram Rajamani, R. Govindarajan, Mahmut Kandemir, and the reviewers for their thoughtful feedback.

References

[1] The Parallel Dwarfs Project. `http://paralleldwarfs.codeplex.com`.

[2] F. Aleen and N. Clark. Commutativity Analysis for Software Parallelization: Letting Program Transformations See the Big Picture. In *ASPLOS*, 2009.

[3] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: An Adaptive, Generic Parallel C++ Library. *LNCS*, 2624:195–210, 2003.

[4] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *CACM*, 52(10), 2009.

[5] E. Berger, K. McKinley, R. Blumofe, and P. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *ASPLOS*, 2000.

[6] E. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe Multithreaded Programming for C/C++. In *OOPSLA*, 2009.

[7] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August. Revisiting the sequential programming model for multi-core. In *MICRO*, 2007.

[8] S. Burckhardt, A. Baldassion, and D. Leijen. Concurrent Programming with Revisions and Isolation Types. In *OOPSLA*, 2010.

[9] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. *SIGPLAN Notices*, 2004.

[10] M. Carlisle. *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines*. PhD thesis, Princeton University, June 1996.

[11] T. Chen, F. Min, V. Nagarajan, and R. Gupta. Copy or Discard Execution Model for Speculative Parallelization on Multicores. In *MICRO*, 2008.

[12] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization*. Birkhauser Boston, 1st edition, 2000.

[13] K. Datta. *Auto-tuning Stencil Codes for Cache-Based Multicore Platforms*. PhD thesis, UC Berkeley, December 2009.

[14] R. Dias, J. Seco, and J. Lourenco. Snapshot isolation anomalies detection in software transactional memory. In *INForum*, 2010.

[15] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software Behavior Oriented Parallelization. In *PLDI*, 2007.

[16] R. Guerraoui and M. Kapalka. On the Correctness of Transactional Memory. In *PPoPP*, 2008.

[17] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S. Liao, E. Bugnion, and M. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, 29:84–89, 1996.

[18] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. In *ISCA*, 1993.

[19] W. Hwu, S. Ryoo, S. Ueng, J. Kelm, I. Gelado, S. Stone, R. Kidd, S. Baghsorkhi, A. Mahesri, S. Tsao, N. Navarro, S. Lumetta, M. Frank, and S. Patel. Implicitly parallel programming models for thousand-core microprocessors. In *DAC*, 2007.

[20] K. Kennedy and J. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., 2002.

[21] V. Krishnan and J. Torrellas. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Trans. on Computers*, 48(9), 1999.

[22] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali. Lonestar: A Suite of Parallel Irregular Programs. In *ISPASS*, 2009.

[23] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Optimistic Parallelism Benefits from Data Partitioning. In *ASPLOS*, 2008.

[24] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic Parallelism Requires Abstractions. In *PLDI*, 2007.

[25] J. Larus and R. Rajwar. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. Morgan & Claypool Publishers, 2007.

[26] D. Maydan, S. Amarasinghe, and M. Lam. Array data-flow analysis and its use in array privatization. In *POPL*, 1993.

[27] M. Mehrara, J. Hao, P. Hsu, and S. Mahlke. Parallelizing Sequential applications on Commodity Hardware using a Low-cost Software Transactional Memory. In *PLDI*, 2009.

[28] C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *IISWC*, 2008.

[29] S. Misailovic, D. Kim, and M. Rinard. Parallelizing Sequential Programs With Statistical Accuracy Tests. Technical Report MIT-CSAIL-TR-2010-038, MIT, Aug 2010.

[30] S. Moon and M. W. Hall. Evaluation of Predicated Array Data-flow Analysis for Automatic Parallelization. In *PPoPP*, 1999.

[31] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in C*. Cambridge University Press, 2nd edition, 1992.

[32] W. Pugh and D. Wonnacott. Constraint-based array dependence analysis. *ACM TOPLAS.*, 1998.

[33] E. Raman, N. Vachharajani, R. Rangan, and D. August. SPICE: Speculative Parallel Iteration Chunk Execution. In *CGO*, 2008.

[34] L. Rauchwerger and D. Padua. The Privatizing DOALL Test: A Run-Time Technique for DOALL Loop Identification and Array Privatization. In *ICS*, 1994.

[35] L. Rauchwerger and D. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. In *PLDI*, 1995.

[36] T. Reps. Undecidability of context-sensitive data-independence analysis. *ACM TOPLAS*, 2000.

[37] T. Riegel, C. Fetzer, and P. Felber. Snapshot isolation for software transactional memory. In *TRANSACT*, 2006.

[38] M. Rinard and P. Diniz. Commutativity Analysis: A New Analysis Technique for Parallelizing Compilers. *ACM TOPLAS*, 19(6), 1997.

[39] G. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *ISCA*, 1995.

[40] R. Tarjan. A Unified Approach to Path Problems. *J. ACM*, 28(3), 1981.

[41] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs. In *MICRO*, 2007.

[42] C. Tian, M. Feng, and R. Gupta. Supporting Speculative Parallelization in the Presence of Dynamic Data Structures. In *PLDI*, 2010.

[43] G. Tournavitis, Z. Wang, B. Franke, and M. O'Boyle. Towards a Holistic Approach to Auto-parallelization: Integrating Profile-driven Parallelism Detection and Machine-learning Based Mapping. *PLDI*, 2009.

[44] N. Vachharajani, R. Rangan, E. Raman, M. Bridges, G. Ottoni, and D. August. Speculative Decoupled Software Pipelining. In *PACT*, 2007.

[45] H. Vandierendonck, S. Rul, and K. Bosschere. The Paralax infrastructure: automatic parallelization with a helping hand. In *PACT*, 2010.