Towards the Analysis of Transactional Software

Abstract

The computer-architecture community's recent focus on multi-core architectures has spurred renewed interest in concurrent-programming techniques and abstractions. For programmers to take advantage of the processing power of today's multi-core chips, they need to write multi-threaded applications. Specifically, the programming-language community has focused on software transactions. A software transaction declaratively specifies what program statements should execute atomically. Our research focuses on the analysis of programs that make use of software transactions. We present a novel interprocedural analysis, XRef analysis, that annotates the fields of each record type in the program with the static transactions in which it is referenced. We show how the results of XRef analysis can be used to perform additional analyses. In particular, we present two such analyses, XOrder and XProtect. XOrder is aimed at software running on a transactional-memory platform and attempts to find an optimized layout for a record in memory. XProtect is a safety analysis that warns the programmer if a shared record is inconsistently protected by a software transaction.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Optimization; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Progamming Languages—Program analysis

1. Introduction

Software developers have traditionally benefited from advances in computer architecture, which, over multiple decades, has provided a steady increase in the computing resources available to applications. However, the immense heat dissipation and power usage that results from ramping up the clock frequency has (temporarily) stalled this process.

To continue to provide software developers with increased computing resources, computer architecture has focused on thread-level parallelism. This is evident by the myriad of chip multiprocessors (CMP) in today's commodity PCs. A CMP consists of multiple processing cores on one die. Many of today's laptops now include dual-core processors. There are predictions that the next generation of processors will be 4 or 8-way CMPs.

The CMP trend in computer architecture implies that software developers will need to write concurrent programs to take advantage of the increased processing power. In the past, the synchronization primitives used to write concurrent programs have traditionally been locks, semaphores, and condition variables. These primitives are both error-prone and do not scale to many threads of execution. Furthermore, it has been shown that this is neither composable nor effective for large data structures, such as hashtables [17].

The difficulties of lock-based programming led to research in programming with *transactions*. In abstract terms, a transaction is a sequence of events that appear to happen atomically. The idea of using transactions was borrowed from the database community. Traditional database transactions support ACID (Atomicity, Consistency, Isolation, and Durability) semantics. To avoid confusion, we will refer to transactions used in programs as software transactions.

A *software transaction* consists of a sequence of instructions executed by one thread, whose side effects appear to happen at once. Software transactions give programmers a declarative way of expressing synchronization—the programmer will specify *what* program statements are to be executed atomically and not how the atomic execution is implemented. Due to the volatile nature of memory, software transactions only concern themselves with the first three principles of database transactions, namely ACI.

Software transactions require the support of a *transactional memory* (TM) system. In order to support concurrency, TM systems attempt to execute transactions concurrently, but enforce atomicity and isolation by stalling or aborting transactions that conflict. A *conflict* occurs when two transactions access the same data concurrently, and one of them is a write. There have been many research proposals, from both the architecture and programming languages communities, on the implementation of transactional memory. These include transactional-memory systems that are implemented completely in software (STM) [38, 16, 31, 19], in hardware (HTM) [20, 15, 2, 28, 24], and implementations that have various aspects implemented in hardware and software (HyTM) [12, 21].

Software transactions strive to facilitate the writing of correct highly-concurrent programs. Programmers can leave the implementation of atomic critical sections to the underlying system. Improper use of locks, for example, can lead to deadlock. By the abstract specification of atomic sections, performance of multi-threaded programs can be improved by executing software transactions that operate on disjoint data in parallel. Lock-based critical sections, on the other hand, are serialized if they are protected by the same lock regardless of the data actually accessed.

In anticipation of this new programming paradigm, we introduce XRef, a novel interprocedural static analysis that annotates the fields of each record type[1] in the program with the static transactions (Defn. 4.3) in which they are referenced. These (automatically inferred) annotations are important for analysis tools whose focus includes transactional software. That is, the results of XRef analysis form a basic building block on which other analyses can be built. We demonstrate this capability by implementing two other interprocedural analyses, XOrder and XProtect, on top of XRef.

[1] We use the term "record" to emphasize that the analysis is language independent. A record type corresponds to a C `struct` or to a class in an object-oriented language.

While software transactions present a step forward in the evolution of concurrent programming, improper use of transactions can erode their benefits. A program's performance can suffer if concurrency is limited by conflicting transactions. The problem of transaction conflicts can be exacerbated by transactional memory systems that detect conflicts at a coarse granularity. For example, many HTM's detect conflicts between concurrent transactions using the cache coherence mechanism. To conserve hardware resources, these systems typically detect conflicts at the granularity of a cache line. In such systems, concurrent accesses to different words in the same cache line by two different transactions can cause a *false conflict* a unnecessary serialization of transactions that is due solely to the heuristic nature of a platform's conflict-detection mechanism—see Defn. 6.1). False conflicts can cause significant performance loss on HTMs [24]. We address false conflicts through XOrder analysis. Specifically, the goal of XOrder analysis is to reduce false conflicts among concurrent software transactions. XOrder analysis targets TM systems systems that detect conflicts at a coarse granularity. Using a detailed simulator of an HTM system, we measure the speedups enabled by XOrder; these ranged from **MISSING** to **MISSING**.

As with performance problems, programmers using software transactions must still reason about program correctness. For example, bugs may result from data races on shared data structures that are inconsistently protected by transactions. We address this problem with XProtect analysis. XProtect verifies data-race freedom by ensuring that all accesses to a shared instance of a global record are consistently protected by a software transaction.

The contributions of this paper are:

- XRef, a new software analysis that annotates the fields of a record with the static transactions in which they are accessed.
- XOrder, a software optimization that determines a memory layout for the fields of a record. This optimization is for transactional software running on a hardware implementation of transactional memory.
- XProtect, a safety analysis that warns the programmer when an access to a shared structure may not be protected by a transaction.
- Our analyses are able to analyze existing lock-based programs in a transactional setting.
- Our experiments show that we can perform these analyses efficiently. **MISSING. We measured speedups of x - y %**

The remainder of the paper is organized as follows: §2 specifies the programming model. §3 and §4 introduce the notion of a dynamic transaction and static transaction, respectively. §5 presents XRef. §6 and §7 present our additional program analyses XOrder and XProtect, respectively. §8 discusses related work.

2. Programming Model

Our programming model consists of a multithreaded program running on a platform that supports transactional memory. Transactional memory may be implemented in hardware, software, or a combination of the two. Although the underlying implementation does not affect our analyses, in this paper, we focus on transactional memory systems that detect conflicts at a granularity larger than that of a memory address—for instance, at the level of a cache line. Detecting conflicts at cache-line granularity is common in HTM systems that perform conflict detection through the cache-coherence protocol [2, 20, 24, 28], in Hybrid TM systems [12, 21], and in at least one STM implementation [33]. The transactional memory system's atomicity model, either strong or weak [3], has no effect on our analyses.

Program	$\rightarrow$	*RecordList FunList*
RecordList	$\rightarrow$	ϵ | *RecordList Record*
Record	$\rightarrow$	record r_{id} $\{t{:}id_1, \ldots, t{:}id_k\}$
FunList	$\rightarrow$	ϵ | *FunList Fun*
Fun	$\rightarrow$	t fun f_{id} $(t{:}p_1, \ldots, t{:}p_k)$ { *Stmt* }
Stmt	$\rightarrow$	$t\,x$ | x = *Exp* | *Stmt;Stmt*
	|	if *Exp* then *Exp* else *Exp*
	|	while *Exp* { *Stmt* }
	|	return *Exp*
	|	begin_transaction
	|	end_transaction
	|	thread { *Stmt* }
Exp	$\rightarrow$	n | x | $x.id_j$ | *Exp op Exp*
	|	$f_{id}(Exp_1, \ldots, Exp_k)$
op	$\rightarrow$	+ | − | * | / | *lt* | *gt* | *eq* | *or* | *and*
t	$\rightarrow$	*int* | *boolean* | r_{id} | *void*

Figure 1. The source language.

```
record ColoredShape { int:r, int:g, int:b,
               int:x1, int:y1, int:x2, int:y2,
               int:x3, int:y3, int:x4, int:y4
}
void fun muly(ColoredShape:s, int amt) {
    s.y1 = s.y1 * amt; s.y2 = s.y2 * amt;
    s.y3 = s.y3 * amt; s.y4 = s.y4 * amt;
}
void fun mulx(ColoredShape:s, int amt) {
n1: begin_transaction;
    s.x1 = s.x1 * amt; s.x2 = s.x2 * amt;
    s.x3 = s.x3 * amt; s.x4 = s.x4 * amt;
}
int sumx(ColoredShape:s) {
    int tot = s.x1 + s.x2 + s.x3 + s.x4;
    return tot;
}
int fun main() {
    ColoredShape s = ...;
n2: thread { begin_transaction; muly(s,3);
n3:          end_transaction; }
n4: thread { mulx(s,0); end_transaction; }
n5: thread { int xtot = sumx(s); ... }
}
```

Figure 2. Example program. Labels n1-n5 will be used to explain various aspects of the analyses.

For expository purposes, we use the simplified programming language specified by the grammar in Fig. 1. In the specification, n is an integer, x is an identifier, and t is a type. Fig. 1 defines a C-like language that includes support for Booleans, integers, records, variables, named functions, threads, and software transactions. All records are passed by reference. Fig. 2 shows an example program. Although it does not perform any meaningful task, it is intended to help clarify the capabilities of XRef, XOrder, and XProtect.

A software transaction is started by the `begin_transaction` statement and completed by the `end_transaction` statement. Our model differs from past research, which uses an `atomic`-block construct, by permitting the beginning and ending of a software transaction to be specified separately. This choice is a direct result of the authors' desire to implement and experiment with the analyses that will be discussed later. That is, the number of benchmarks available today that make use of the `atomic`-block construct is currently very small. Therefore, our experiments analyze lock-based programs by treating a lock aquire as a `begin_transaction` and a lock release as an `end_transaction`.[2] In one of the benchmarks discussed in §6.1, the two ends of a matched `begin_transaction` and `end_transaction` pair occur in separate files. Hence, we needed a programming model that could handle the case where a program begins a transaction in one function and ends it in another.

We do expect future programming languages to use the `atomic`-block construct. Because of this, we present two versions for each of our analyses. The first version is for the analysis of programs that use the `atomic`-block construct, while the second is for programs that adhere to the less-structured programming model. Our goal is to allow researchers to perform static analysis of software in a transactional setting. By presenting both versions, we provide algorithms for analyzing software that makes use of an `atomic`-block construct, as well as for analyzing lock-based programs in a transactional setting. With respect to analyzing lock-based programs, the algorithms we present are useful for both analyzing current lock-based programs for execution on research TM implementations, as well as for analyzing legacy code when TM has become standard.

3. Dynamic Transaction

Here we define the notion of a dynamic transaction. The purpose of this definition is to make clear the distinction between dynamic and static transactions (Defn. 4.3). Dynamic transactions are a runtime notion describing an actual execution of a program, whereas static transactions are intrinsic to the program's text. We present two definitions of a dynamic transaction: (1) for programs that use `atomic` blocks, and (2) for programs that adhere to the less-structured programming model (we refer to these software transactions as unstructured).

DEFINITION 3.1. *A dynamic transaction is a sequence of instructions executed in the scope of an* `atomic` *block.*

DEFINITION 3.2. *A dynamic transaction is a sequence of instructions executed between a matched pair of* `begin_transaction` *and* `end_transaction` *instructions.*

4. Static Transactions

A program-analysis tool for understanding transactional software must be able to distinguish between program statements that occur within a dynamic transaction and those that do not. Because a dynamic transaction is a runtime notion, a static-analysis tool cannot work with dynamic transactions directly. For the static analysis of transactional software, we introduce the notion of a *static transaction*. A static transaction is defined in terms of *valid paths* in the program.

DEFINITION 4.1. *A path in a program's interprocedural control-flow graph (ICFG) is valid if it respects the fact that when a procedure finishes, it returns to the site of the most recent call [37].*

Note that a given valid path may not actually be executable, e.g., due to correlations in the path's branch-conditions. As we did for dynamic transactions, we present two definitions of a static transaction: (1) for programs that make use of atomic blocks, and (2) for programs that adhere to our programming model.

DEFINITION 4.2. *A static transaction consists of all program statements that lie on a valid path within an* `atomic` *block. A static transaction is represented by the program locations that demarcate the block.*

DEFINITION 4.3. *A static transaction consists of all program statements that lie on a valid path between a matched pair of* `begin_transaction` *and* `end_transaction` *statements. A static transaction is represented by the pair of program locations for the* `begin_transaction` *and* `end_transaction` *statements.*

A static transaction represents a set of dynamic transactions, namely, those that start at the `begin_transaction` point and end at the `end_transaction` point. Our analyses use sets of static transactions to over-approximate sets of dynamic transactions.

4.1 Atomic Blocks

It is trivial to determine a program's static transactions when the programming language supports `atomic` blocks. In standard compiler infrastructures, the parser will check for correct syntax of the program and return an abstract syntax tree (AST). The parser can be augmented to record the location of each atomic block in the program. With this information, the node set of a static transaction consists of all program nodes that are within the scope of its `atomic` block and the program nodes of all functions (transitively) called from within that scope.

4.2 Unstructured Software Transactions

To analyze traditional lock-based code in a transactional setting, we treat each lock-acquire and lock-release operation as a `begin_transaction` and `end_transaction`, respectively [27, 6]. This adds considerable difficulty to determining the static transactions of the program—the parser no longer accomplishes this. Not only that, but also the parser no longer verifies correct usage of the `begin_transaction` and `end_transaction` primitives. In this section, we present an interprocedural dataflow analysis to address these problems. Namely, the analysis (1) verifies correct usage of the transactional primitives, and (2) determines the static transactions of a program.

Because converted lock-based programs are allowed to specify the beginning and ending of a software transaction separately, we must take care not to make the detection of static transactions undecidable. The problem is that we are close to having to contend with two interleaved sets of matched parentheses, one (valid paths) for modeling the control flow of the program, and one for modeling the matched `begin_transaction` and `end_transaction` statements. Reps [30] proved that a program-analysis problem that must meet the matching constraints of two interleaved (i.e., shuffled) sets of matched parentheses is undecidable in the general case. To sidestep this undecidability result, we restrict our programming model to allow transactions to be nested to only a finite depth. For the purposes of presenting our analyses, we set the nesting depth to be zero. That is, we will not consider nested transactions in this paper. With a finite nesting depth, we can verify correct usage of the software-transaction primitives using a property automaton [14, 4]. The property automaton for checking correct usage of the `begin_transaction` and `end_transaction` primitives with nesting depth of zero is shown in Fig. 3. It is trivial to expand this automaton to allow for any finite number of nested transactions.

Property automata are useful for answering reachability questions, like incorrect usage of the transactional primitives. This is often done by creating a dataflow-analysis problem in which the automaton's transition relation (often denoted by δ) is used as a state transformer. (More precisely, each transformer is δ restricted to one or more alphabet symbols.) Tab. 1 shows the transformers that model the property automaton from Fig. 3 (for now ignore the bracketed symbols in the subscripts). In Tab. 1, the transformer

[2] Blundell et al. showed that it is possible to introduce deadlocks when lock-based programs are run/analyzed transactionally by converting lock acquire and lock release into `begin_transaction` and `end_transaction`, respectively [3]. An analysis that detects such situations is not the focus of this paper.

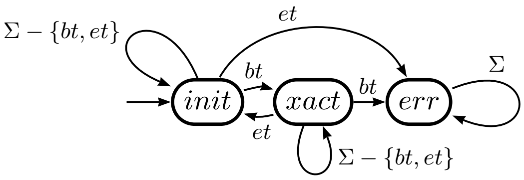

Figure 3. Property automaton for validating correct transaction usage with a nesting depth of 0. In the diagram, *bt* represents `begin_transaction`, *et* represents `end_transaction`, and Σ represents all symbols.

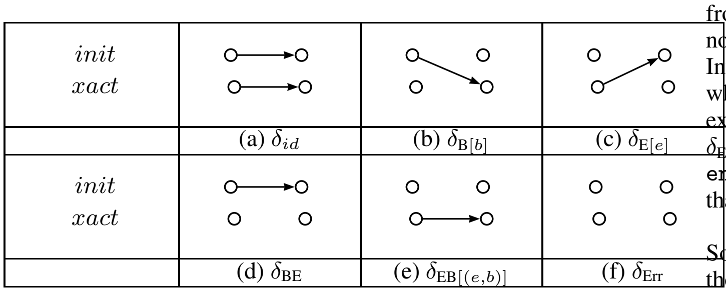

Table 1. δ-transformers that encode the transition relation of the property automaton shown in Fig. 3. The left column is a legend labeling the states of the δ-transformers.

states model the states of the property automaton. To reduce clutter, transitions to the err state are not modeled explicitly; instead, they are modeled by a state q on the left-hand side being disconnected from every state on the right-hand side. This represents an implicit transition from q to the err state. In Tab. 1, the top row displays the transformers that correspond to the program statements. Transformer $\delta_{\mathrm{B}[b]}$ corresponds to a `begin_transaction` statement, transformer $\delta_{\mathrm{E}[e]}$ corresponds to an `end_transaction` statement, and transformer δ_{id} corresponds to all other program statements. The second row shows the transformers that can arise through transformer composition.

To verify correct usage of the `begin_transaction` and `end_transaction` primitives, we solve an interprocedural dataflow-analysis problem. We associate dataflow transformers with each edge in the ICFG. To make clear the distinction between dataflow transformers and transformers that model the transition relation of an automaton, we will refer to the latter as δ-transformers, and use the symbol τ for dataflow transformers.

To determine correct usage of the transactional primitives, our dataflow-analysis transformers (τ-transformers) are sets of δ-transformers that model the property automaton. Associated with each edge in the ICFG is a singleton set containing the δ-transformer that encodes the effect of abstractly executing that edge (with respect to the property automaton). Composition of two transformers τ_1 and τ_2 involves taking the cross product of the two sets of δ-transformers, and performing relational composition on the resulting pairs of δ-transformers. We denote relational composition by ";" (e.g., $\delta_{id}; \delta_{\mathrm{B}[b]} = \delta_{\mathrm{B}[b]}$ and $\delta_{\mathrm{B}[b]}; \delta_{\mathrm{E}[e]} = \delta_{\mathrm{BE}}$).

To solve the dataflow-analysis problem, we use standard techniques [37] to compute a fixpoint. The result of this computation is a set of δ-transformers for each node in the ICFG, where a node's set contains the meet-over-all-paths (MOP) value along valid paths from program entry to that node. To check for correct usage of the transactional primitives, one need only verify that $\tau_{\text{exit}} \subseteq \{\delta_{id}, \delta_{\mathrm{BE}}\}$, where τ_{exit} is the dataflow transformer associated with the exit node of the `main` function.

We extend the τ-transformers to collect additional information about the program. Sagiv et al. showed how relations can be annotated with "values" that satisfy certain algebraic properties [32]. Since our dataflow transformers are sets of relations on the states of the property automaton, we use this framework to provide a more general framework for analyzing programs that make use of software transactions (in a manner similar to our programming model). Our first usage of this framework is to determine all of the static transactions of a program.

Let B and E be the set of all `begin_transaction` and `end_transaction` program locations, respectively. We annotate certain δ-transformers with either an element in B or E or a tuple from $E \times B$. Revisiting Tab. 1, the δ-transformers that contain annotations are marked by the bracketed symbols in their subscript. Intuitively, the annotation on a δ-transformer is an "explanation" why that δ-transformer does not contain the edge $init \rightarrow init$. For example, the δ-transformer $\delta_{\mathrm{EB}[(e,b)]}$ in table entry (e) arises from $\delta_{\mathrm{E}[e]}; \delta_{\mathrm{B}[b]}$. It is annotated with (e, b), which records the unmatched `end_transaction` and `begin_transaction` program statements that caused this δ-transformer to arise.

With these annotated δ-transformers, Tab. 2 defines $\delta_1; \delta_2$. Some of the entries in Tab. 2 are highlighted. This is because for those entries, the composition of two annotated δ-transformers also produces an auxiliary tuple (b, e), where $b \in B$ and $e \in E$, which is placed in a global database. These tuples allow us to determine the static transactions of a program.

To use our dataflow transformers in the framework defined in [32], we must show that they adhere to certain algebraic properties. This requires proving that our dataflow transformers form a finite-height meet semi-lattice. Semi-lattice elements are sets of δ-transformers. Let Δ be the set of all δ-transformers. The semi-lattice is finite-height because there is only a finite number of δ-transformer sets, i.e., $\mathcal{P}(\Delta)$. We define the meet operation, $\sqcap$, as set union. Semi-lattice elements are ordered by superset. Because composition of τ-transformers is performed pointwise (i.e., on a per-δ-transformer basis), composition distributes over $\sqcap$; hence, the problem is distributive.

For determining the static transactions of a program, each dataflow transformer is a set of annotated δ-transformers. We associate with each edge in the ICFG the singleton set containing the annotated δ-transformer that corresponds to abstractly executing that edge. Then, using standard techniques [37], the meet-over-all-paths value for each node is found by computing a fixpoint. When the composition of two δ-transformers produces a tuple, we record it in a global database. After the fixpoint has been reached, the database contains all of the program's static transactions. Because we build on the framework that verifies correct usage of the transactional primitives, this analysis also verifies correct usage of the primitives.

Our implementation uses weighted pushdown systems (WPDSs) [35, 29] as the dataflow analysis solver. Reps et al. showed the connection between WPDSs and dataflow analysis [29]. The WPDS algorithms for solving a dataflow-analysis query have the capability to return a *witness trace* [29] for the answer found for each program node. A witness trace can be thought of as a proof for the computed answer. We exploit this capability for alerting the programmer of possible incorrect usage of the `begin_transaction` and `end_transaction` primitives. That is, if after solving the dataflow analysis, $\tau_{\text{exit}} \cap \{\delta_{\mathrm{B}[b]}, \delta_{\mathrm{E}[e]}, \delta_{\mathrm{EB}[(e,b)]}, \delta_{\mathrm{Err}}\} \neq \emptyset$, then there is possible incorrect usage of the primitives. We present to the programmer a path through the program that produces this erroneous result. Due to the abstraction being used, the path may not actually be executable (e.g., due to correlated branches). The programmer can analyze the path and determine if there exists an actual misuse of the primitives in the program.

$\delta_1;\delta_2$	δ_{id}	$\delta_{B[b_2]}$	$\delta_{E[e_2]}$	δ_{BE}	$\delta_{EB[(e_2,b_2)]}$	δ_{Err}
δ_{id}	δ_{id}	$\delta_{B[b_2]}$	$\delta_{E[e_2]}$	δ_{BE}	$\delta_{EB[(e_2,b_2)]}$	δ_{Err}
$\delta_{B[b_1]}$	$\delta_{B[b_1]}$	δ_{Err}	$\delta_{BE}, (b_1, e_2)$	δ_{Err}	$\delta_{B[b_2]}, (b_1, e_2)$	δ_{Err}
$\delta_{E[e_1]}$	$\delta_{E[e_1]}$	$\delta_{EB[(e_1,b_2)]}$	δ_{Err}	$\delta_{E[e_1]}$	δ_{Err}	δ_{Err}
δ_{BE}	δ_{BE}	$\delta_{B[b_2]}$	δ_{Err}	δ_{BE}	δ_{Err}	δ_{Err}
$\delta_{EB[(e_1,b_1)]}$	$\delta_{EB[(e_1,b_1)]}$	δ_{Err}	$\delta_{E[e_1]}, (b_1, e_2)$	δ_{Err}	$\delta_{EB[(e_1,b_2)]}, (b_1, e_2)$	δ_{Err}
δ_{Err}	δ_{Err}	δ_{Err}	δ_{Err}	δ_{Err}	δ_{Err}	δ_{Err}

Table 2. Definition of composition of δ-transformers. The labels in square brackets represent the annotation on a δ-transformer. Light grey table entries contain a δ-transformer and a pair, which denotes that the composition also produces a static transaction.

5. XRef

It is useful for a programmer to know what record fields are accessed in the dynamic transactions during program execution. This is important both for debugging and for understanding a large unfamiliar code base. For example, when adding fields to a record type or splitting a large record type into two, the programmer must be aware of the fact that the fields of an instance of that record are guarded by a software transaction. We are able to present to the programmer an approximation of this information through XRef analysis. XRef analysis approximates the set of fields accessed in dynamic transactions by annotating the fields of each record type with the static transactions in which they are referenced. XRef analysis can be viewed as inferring annotated types, where the annotations are the static transactions in which each field of a record type is referenced.

XRef analysis does not distinguish between different instances of a record type. Later, in §7, we present an analysis that is capable of attaching different annotations to different instances of a record type.

5.1 Atomic Blocks

When the program being analyzed makes use of the `atomic`-block construct, XRef analysis can be performed by a flow-insensitive interprocedural analysis (similar to GMOD/GUSE analysis [8]). Because of the similiarities between XRef analysis (for the `atomic`-block construct) and GMOD/GUSE analysis, we only sketch the algorithm.

Let R be the set of all record-type names, and F be the set of all field names. For each static transaction, the algorithm computes a set of pairs s, where each pair is of the form (r, f) with $r \in R$ and $f \in F$. For each function of the program, perform an intraprocedural analysis that determines all record-type field accesses performed by that function. The accesses are represented by a set of pairs s. This can be accomplished with a single pass over each program node in a function's control-flow graph (similar to IUSE analysis). Once this information has been computed, next perform a fixpoint calculation over the callgraph to determine the field accesses made transitively by each function (similar to GUSE analysis). Finally, for each function that contains an `atomic` block, traverse the program statements that are in the scope of the `atomic` block, unioning the accesses for each program statement. If a program statement is a function call, the set of accesses is retrieved from the sets that were calculated from the fixpoint computation on the callgraph. Once this information has been computed for each static transaction, the fields of each record type are annotated with the static transaction(s) in which it is accessed.

5.2 Unstructured Software Transactions

We make use of the framework presented in §4.2 to analyze lock-based programs converted to use `begin_transaction` and `end_transaction`. To do this, we add to each δ-transformer's annotation a set of pairs, as defined in §5.1. Specifically, for each

Program Statement	δ-transformer	Pair Set
n1: `begin_transaction`	$\delta_{B[n1]}$	$\emptyset$
n1: `end_transaction`	$\delta_{E[n1]}$	$\emptyset$
n1: $x = r_1.f_1$	δ_{id}	$\{(r_1, f_1)\}$
n1: $r_1.f_1 = x$	δ_{id}	$\{(r_1, f_1)\}$

Table 3. Example program statements; their corresponding δ-transformers; and the set of pairs that annotates them. annotations.

δ-transformer in the top row of Tab. 1, we add to the annotation a set of pairs s, which is shown in Tab. 3. We make a few changes to the composition of δ-transformers. For brevity, we enumerate the changes here instead of replicating Tab. 1. We use the notation (s, δ) to represent an annotated δ-transformer extended with the set of pairs s. Likewise, we use the notation $(s_1, \delta_{EB[(e,b)]}, s_2)$ to represent that $\delta_{EB[(e,b)]}$ has two sets, s_1 and s_2. We define ";" as follows (see below for an explanation):

1. $(s_1, \delta_{B[b]}); (s_2, \delta_{E[e]}) = (\emptyset, \delta_{BE}), (b, e, s_1 \cup s_2)$
2. $(s_1, \delta_{id}); (s_2, \delta_{B[b]}) = (s_2, \delta_{B[b]})$
3. $(s_1, \delta_{BE}); (s_2, \delta_{B[b]}) = (s_2, \delta_{B[b]})$
4. $(s_1, \delta_{E[e]}); (s_2, \delta_{B[b]}) = (s_1, \delta_{EB[(e,b)]}, s_2)$
5. $(s_1, \delta_{E[e]}); (s_2, \delta_{id}) = (s_1, \delta_{E[e]})$
6. $(s_1, \delta_{E[e]}); (s_2, \delta_{BE}) = (s_1, \delta_{E[e]})$
7. $(s_1, \delta_{id}); (s_2, \delta_{EB[(e_1,b_1)]}, s_3) = (s_1 \cup s_2, \delta_{EB[(e_1,b_1]}, s_3)$
8. $(s_1, \delta_{EB[(e_1,b_1)]}, s_2); (s_3, \delta_{id}) = (s_1, \delta_{EB[(e_1,b_1]}, s_2 \cup s_3)$
9. $(s_1, \delta_{B[b_1]}); (s_2, \delta_{EB[(e_2,b_2)]}, s_3) = (s_3, \delta_{B[b_2]}), (b_1, e_2, s_1 \cup s_2)$
10. $(s_1, \delta_{EB[(e_1,b_1)]}, s_2); (s_3, \delta_{E[e_2]}) = (s_1, \delta_{E[e_1]}), (b_1, e_2, s_2 \cup s_3)$
11. $(s_1, \delta_{EB[(e_1,b_1)]}, s_2); (s_3, \delta_{EB[(e_2,b_2)]}, s_4) = (s_1, \delta_{EB[(e_1,b_2]}, s_4), (b_1, e_2, s_2 \cup s_3)$
12. otherwise $(s_1, \delta_1); (s_2, \delta_2) = (s_1 \cup s_2, (\delta_1; \delta_2))$

Let us now explain this definition by means of a few examples. Intuitively, we want to only record field accesses that occur within a static transaction. This is exemplified by rule (1). The composition $(s_1, \delta_{id}); (s_2, \delta_{B[b]})$ results in $(s_2, \delta_{B[b]})$. This has the effect that the composition ignores all field accesses that occur *before* the `begin_transaction` statement. Similiarly, rules (3) and (4) ignore all field accesses that occur *after* an `end_transaction` statement. Notice that the cases specified in rules (6) and (7) each produce an auxiliary triple. This is because those rules result from the matching of a `begin_transaction` and `end_transaction` statement. Specifically, for rule (6), the composition of $(s_1, \delta_{EB[(e_1,b_1)]}, s_2); (s_3, \delta_{E[e_2]})$ results in the annotated δ-transformer $(s_1, \delta_{E[e_1]})$ along with the auxiliary triple $(b_1, e_2, s_2 \cup s_3)$. The triple denotes that static transaction (b_1, e_2) accessed the fields contained in the set $s_2 \cup s_3$. When an auxiliary triple is produced, we record this information in a global database.

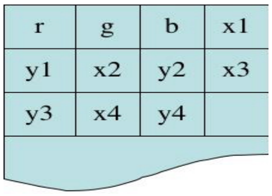

Figure 4. Possible layout in memory of the shared record `s` from Fig. 2. Each row represents one cache line of memory.

Each node of the ICFG is annotated with its corresponding dataflow transformer as in §4.2. As before, we use standard techniques to compute the MOP dataflow transformer for each program node in the ICFG. Once the dataflow analysis has completed, the global database contains triples of the form (b, e, s), such that $b \in B$, $e \in E$, and s is a set of record-type-field accesses. We use this information to annotate the fields of each record type with the static transactions in which it is accessed.

When interpreting the results our experiments, we found XRef analysis to be extremely helpful. Indeed, analyzing the information obtained from XRef analysis led us to realize that dynamic transactions were aborting because of false conflicts. We address this issue with our next analysis, XOrder.

6. XOrder

XOrder is an analysis that determines an improved layout for the fields of a record type. It identifies these reorderings by analyzing the results of XRef analysis. The experiments reported in §6.1, show that, without programmer intervention, XOrder can reduce the rate of false conflicts for programs running on a transactional memory system.

DEFINITION 6.1. *A false conflict occurs when two distinct dynamic transactions access separate bytes of memory, but, due to the conflict-detection scheme of the transactional memory system, one of them is forced to stall or abort.*

Although false conflicts can occur in both STMs and HTMs, they are especially prevelent in HTMs that detect conflicts using the coherence mechanism. For clarity, in this section, we assume an HTM.

In our target HTM implementation, the granularity of precision is one cache line. Therefore, during a dynamic transaction, the HTM's conflict-detection mechanism does not distinguish between memory reads and writes to the same cache line. Fig. 4 depicts a possible layout in memory of the shared `ColoredShape` record `s` from the program in Fig. 2. Assuming that integers occupy four bytes of memory and that the cache-line size is sixteen bytes, accesses to any of the fields {`r, g, b, x1`} will be treated equally by the HTM's conflict-detection mechanism. Similarly, accesses to any of {`y1, x2, y2, x3`} will not be distinguished and accesses to any of {`y3, x4, y4`} will not be distinguished. This memory alignment, combined with the HTM's conflict-detection mechanism, will cause a false conflict between the concurrent execution of transactions (`n2`,`n3`) and (`n1`,`n4`) from the program in Fig. 2. It is a false conflict because the two transactions do not access any common fields of the shared `ColoredShape` record `s`. To tackle this problem, we developed an analysis algorithm, XOrder, that determines an optimized layout ordering for record types.

MISSING - reword Record-layout reordering is a technique that analyzes which fields of a particular record type might be accessed in a dynanic transaction. The result of XRef is the set of fields of record types accessed in a static transaction. Thus, XRef computes a superset of the fields actually accessed in a dynamic transaction. The idea behind record-layout reordering is that if two fields, f_1 and f_2, are accessed only in separate dynamic transactions, then the layout of the record type in memory should be chosen to ensure that these fields reside in separate cache lines. Because we are performing static analysis, we do not know where in memory a record instance will be dynamically created, and furthermore we do not know its alignment with respect to cache lines. Therefore, our goals are: (1) to separate f_1 and f_2 with fields that are not accessed in a static transaction, and (2) to ensure that the sum of the sizes of these "padding" fields is greater than or equal to the size of a cache line. A beneficial byproduct of this ordering is that the number of cache lines accessed by a dynamic transaction is potentially decreased. This is akin to the *cache-block working set* optimization described by Chilimbi et al. in [5].

XOrder uses the field annotations inferred by XRef to determine an optimized ordering of a record type's fields. With respect to [5], we do not use dynamic profiling information to determine a field ordering. Instead, we rely on the field accesses that occur in a static transaction.

Intuitively, the ordering determined by XOrder should be such that each static transaction accesses a contiguous region of memory. When the fields accessed by a static transaction do not overlap with the fields accessed by any other static transaction, determining this ordering is trivial. We create an undirected graph, where the nodes of the graph correspond to the fields of the record type. There is an edge between two nodes (fields) if they are accessed in the same static transaction. In Fig. 5, the `x` fields are all connected because they are accessed in the static transaction (`n1`,`n4`) and the `y` fields are all connected because they are accessed in the static transaction (`n2`,`n3`). When ordering these fields, each connected component represents nodes that should be laid out near each other. A (linearization of a) spanning tree for the connected component gives a linear order for those fields.

In some cases, the accesses of two static transactions will share some subset of a record type's fields. We refer to these static transactions as a transactional group.

DEFINITION 6.2. *A transactional group consists of 2 or more static transactions such that there is a common field shared between at least two of the static transactions in the group.*

In these situations, the fields of a transactional group are ordered such that each static transaction of the group accesses contiguous regions of memory. One solution to find such an ordering (if it exists) is to solve an instance of the traveling-salesman problem (TSP). Create a graph where the nodes of the graph represent the fields of the transactional group. For each pair of nodes, add a weighted edge to the graph. The weight of the edge is determined by the static transactions of the group. That is, if the two fields represented by the nodes are accessed in the same static transaction, then the weight is 1. Otherwise, we set the weight to be greater than 1. By setting the weight to be greater than 1, a solution to the TSP will not traverse that edge unless it is absolutely necessary. In our experiments, we used the weight 10. For such graphs, an optimal solution to the TSP will produce the desired ordering. The path can be used to create the record layout. If it is possible for every static transaction of the group to access contiguous memory, then this ordering will provide it. In cases where no such ordering exists, the ordering produced by solving the generated instance of the TSP will provide a good approximation.

Because we are targeting our analysis for a compiler infrastructure, solving an NP-complete problem like the TSP every time the program is modified might be too costly. Therefore, we use the well-known heuristic of using a minimum-spanning-tree (MST) algorithm to approximate the TSP [9]. Approximating the TSP

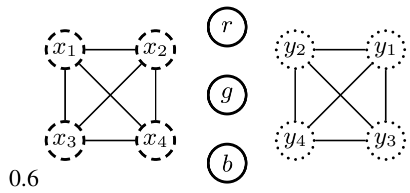

Figure 5. The graph generated during XOrder analysis for the program in Fig. 2.

```
record ColoredShape {
    int:x1, int:x2, int:x3, int:x4,
    int:r, int:g, int:b,/*int:pad1,*/
    int:y1, int:y2,int:y3, int:y4
}
```

Figure 6. A reordering of the fields of ColoredShape suggested by XOrder.

solution using an MST is nice because it still takes the graph's edge weights into account. For example, one could imagine using dynamic-profiling information to adjust the weights on the graph, effectively biasing the solution to the TSP to prefer certain paths (orderings) over others. This would allow XOrder to perform an optimization akin to the *field-layout-affinity* method proposed by Chilimbi et al. [5]. Because we use an MST to approximate a solution to the TSP, we no longer need to distinguish between transactional groups and static transactions (that are not members of a group). This is because we can use a linearization of the MST to compute an ordering for both cases.

Once the layout of each static transaction and transactional group is determined, we next use the record type's fields *not* accessed in a static transaction as padding between these orderings. If there are n ordered sets, then XOrder attempts to find $n - 1$ sets of padding fields such that the size of each set is at least the size of the cache line for the target architecture. To do this, we sort the padding fields in decreasing order by their size and then (attempt to) greedily fill the $n - 1$ sets. Fig. 6 displays the layout of `ColoredShape` after analyzing the program in Fig. 2 with XOrder. Notice that the analysis suggested the insertion of an integer field named `pad1`. This is because for the example program (and target architecture with a cache-line size of 16 bytes), there are not enough fields to place between the two static transactions. Our analysis only suggests the padding in a comment instead of directly emitting it because the addition of extra fields can impact the performance of a program. For example, if the program allocated a million `ColoredShape` objects, then the addition of the padding field would cause 4 MB of memory to be wasted. Additionally, adding pad fields will affect the runtime behavior of the program because it places extra pressure on the cache. Because of the padding suggestions, we do not directly modify the analyzed program. Instead, we emit the reorderings for each record type to the programmer. The programmer can then decide whether to apply the reordering.

6.1 XOrder Experiments

We evaluated XOrder on the Splash2 benchmark suite [41]. The Splash2 programs are written in `C`.[3] We extended CIL [25] to extract a WPDS from the `C` code. Our dataflow transformers are represented with BDDs, using the BuDDy [22] BDD library. XOrder's record layout engine used the `ocamlgraph` library [7] for all graph manipulations.

We ran the programs on the full-system simulator GEMS [23, 40] from the University of Wisconsin-Madison. GEMS simulates the SPARC v9 architecture extended with the LogTM HTM system [24]. The benchmarks were run in both in their original form and after modifying them using the results of XOrder. The experiments were run on a Pentium-4 3.0 GHz machine with 4GB of memory running the Linux 2.6.9 kernel. Tab. 4 presents the results of our experiments.

(TODO. analyze experiments. Fig. 7).

[3] The `C` programming language is neither typesafe nor memory safe. Because of this, both XRef and XOrder are unsound. The application of XRef and XOrder analyses to a type and memory safe language would be sound.

Benchmark	# Procs	Δ-Aborts	% Speedup
raytrace	2	39	99.66
	4	-32	97.82
	8	-17	93.00
	16	16	80.06
	32	-91	35.77
mp3d	2	0	100.00
	4	390	102.44
	8	-682	104.62
	16	12599	117.12
	32	2644	100.00
radiosity	2	14	100.01
	4	-93	97.71
	8	3465	104.76
	16	2005	127.12
	32	2275	110.01

Table 4. Experimental results for running the raytrace, mp3d, and radiosity benchmarks from the Splash2 benchmark suite. Column "Aborts" measures the difference in the number of aborts between running the benchmark after reordering the `C` structs and before. The smaller the value the better. Column "Speedup" is a percenctage of the running time the benchmark took for the layout suggested by XOrder versus the original program. Smaller percentages are better.

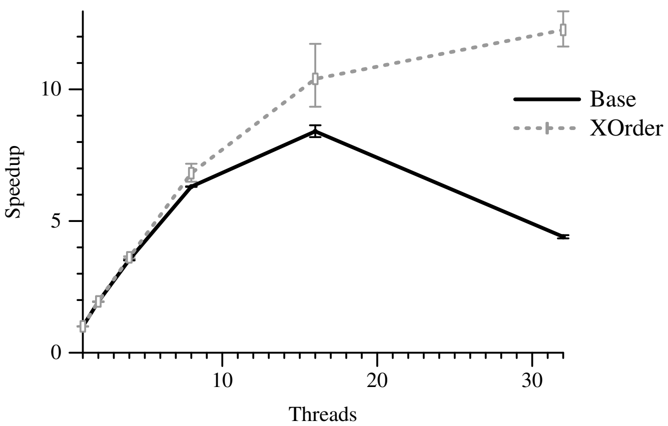

Figure 7. Speedup comparison for the raytrace benchmark. The line labeled "Linear" denotes linear speedup. The lines labeld "Base" and "XOrder" show the speedup before and after optimizing the program with the results of XOrder analysis.

7. XProtect

Software transactions are arguably superior to traditional locks; however, they do not eliminate all programming errors due to concurrency. For example, when used incorrectly, programs using software transactions can still suffer from data races. A data race occurs when two (or more) threads concurrently access the same memory location, and one of them is a write.

In a large software system, the programmer may not know which records are shared between threads and may omit a software transaction. In other cases, the programmer may simply forget to guard a sequence of program statements with a transaction. For instance, in the example program in Fig. 2, the programmer has forgotten to guard the call to `sumx` at location `n5` with a transaction. Consequently, due to the nondeterminism of the thread scheduler, the value returned from that call can have five possible results instead of two: the results depending on when the thread started at location `n4`, t_{n4}, is interleaved with the thread started at location `n5`, t_{n5}. If t_{n4} commits before t_{n5} begins, then the result will be 0. If t_{n4} commits after t_{n5} has read `s.x1`, the result will be `s.x1`. This pattern continues up to the case where t_{n4} commits after t_{n5} has read all of the `x` fields. Had the programmer protected the call to `sumx` with a transaction, there would be only two possible results, namely, (i) 0 in the case that t_{n4} commits before t_{n5}, and (ii) the sum of the values of all the `x` fields otherwise. Similiar problems exist when individual program statements correspond to multiple hardware instructions. Consider the case where one thread transactionally writes a double word of memory (e.g., a `float` in the program). In another thread, the same double word is modified outside of a software transaction. To the underlying system, these writes appear as two separate instructions. Therefore, the underlying TM system is free to place the two writes from the transactional thread between the two writes of the nontransactional thread. This results in the double word containing a partial update from *both* threads.

We address this problem via one more enhancement of our interprocedural static analysis, called XProtect, which builds on the XRef analysis described in §5.

XProtect analysis warns the programmer if accesses to fields of a shared record instance are not consistently protected by a software transaction. Even though this does not guarantee race freedom (e.g., a concurrent program that uses no synchronization is consistent), consistency checking is a common technique for data race detection [26]. We add a sound[4] flow-insensitive, context-insensitive pointer analysis to XRef. Our implementation uses Steensgaard's analysis [39]. We chose this algorithm because it is sound for concurrent programs: a flow-insensitive analysis considers all interleavings of the various threads that make up the concurrent program. Other more expensive pointer analyses that are still sound for concurrent programs would work just as well. We use the term "abstract record-allocation site" to mean the set of record-allocation sites obtained after pointer analysis has finished. As before, we describe XProtect analysis for programs that make use of the `atomic`-block construct and those that come from our programming model. In both cases, we compute the results from two dataflow-analaysis problems, and the answer we are interested in is computed from the differences in the values computed by the two analysis phases.

7.1 Atomic Blocks

For programs that make use of the `atomic`block construct, XProtect is a straightforward extension of XRef. The key is to extend the sets of pairs to include both the abstract record-allocation site that is accessed, as well as an indication of what type of memory-access is made, i.e., either a read or a write. Specifically, we modify the XRef algorithm to compute, for each static transaction, a set of tuples of the form (R, A, F, T), where R and F are as in XRef, A is the set of abstract record-allocation sites obtained from pointer analysis, and $T \in \{read, write\}$. We label each node of the program with a set of tuples s that corresponds to that node's accesses, and the manner in which they are made. For example, a program node that represents the program statement $x = r_1.f_1$ is labeled with the tuple set $\{(r_1.A_{r_1}.f_1.read)\}$, where A_{r_1} is the abstract record-allocation site that is accessed. Using the extended sets of tuples, the algorithm for solving XRef (defined in §5.1) computes, for each abstract record-allocation site, the set of fields either read or written by each static transaction. We denote this set by s_{xact}.

To verify that the field accesses to all shared record instances are consistently protected by a software transaction, we solve a second dataflow-analysis problem, which determines all field accesses that occur outside every static transaction. A second dataflow-analysis problem is required because a function may be called both from within a transaction and outside of one. Therefore, it is possible for that function to make accesses to the same shared record instance in a transactional and nontransactional context.

We collect all nontransactional accesses via a simple trick. We treat each `atomic` block as a "`noop`" instruction. This effectively ignores all static transactions in the program. For each function, we compute the set of field accesses to the abstract record-allocation sites by visiting each node in the function's control flow graph (CFG) and taking the union of their accesses. Next, we solve a set of equations on the program's call-graph to compute the set of abstract record-allocation site field accesses made (transitively) by each function. All nontransactional field accesses to abstract record-allocation sites are precisely the label on the `main` function in the call-graph. We denote this set by s_{access}.

Once s_{access} has been computed, we check to see whether a field is accessed both transactionally and nontransactionally, with one of the accesses being a write. For each read access in s_{xact}, we check to see whether s_{access} contains a write access to the same field. For example, if $(r_1.A_{r_1}.f_1.read) \in s_{\text{xact}}$, which denotes a read of the f_1 field of record type r_1 for abstract record-allocation site A_{r_1}, we check whether $(r_1.A_{r_1}.f_1.write) \in s_{\text{access}}$. Similarly, for each write access in s_{xact}, we check whether there is a corresponding read or write access to the same field in s_{access}. If all checks fail, then the program is data-race free. If either of these checks succeed, then we warn the programmer that there is a potential data race. The warning includes the field and abstract record-allocation site that is inconsistently guarded by a software transaction.

7.2 Unstructured Software Transactions

To perform XProtect analysis on lock-based programs converted to use the `begin_transaction` and `end_transaction` primitives, we extend the XRef analysis presented in §5.2. The sets of pairs used in XRef analysis are extended to be sets of tuples. Each tuple is of the form (R, A, F, T), where R is the set of all record type names, A is the set of abstract record-allocation sites, F is the set of all record-type fields, and $T \in \{read, write\}$. With respect to XRef analysis, the extensions are propagated along. In particular, they have no additional effect when δ-transformers are composed. We note that we only increase the size of the annotations by a finite amount. Moreover, all algebraic properties that were shown to hold earlier are still valid.

As in XRef analysis, we label each edge of the ICFG with its corresponding set of annotated δ-transformers. We then solve the dataflow-analysis problem as before, which populates the global database with the set of fields accessed in a static transaction for each abstract record-allocation site. We deonte the set of abstract

[4] Many pointer analyses are unsound in the presence of uninitialized variables [39, 36]. This can be rectified by also performing uninitialized-variables analysis.

record-allocation site fields accessed in some static transaction by s_{xact}.

To determine the field accesses made to abstract record-allocation sites while *not* in a static transaction, we need to run XProtect again; however, for this second run we modify the definition of ";". The key is to realize that the information sought is the *opposite* of the information computed by XProtect. We compute these unprotected accesses by "inverting" the operations on the annotations of the δ-transformers when composing δ-transformers. Intuitively, we modify the definition of ";" to be such that it only records accesses that occur *outside* of a static transaction. This can be accomplished by making the following changes to ";" (we use the same notation as in §5.2). For instance, in rule (1) below, the tuples in s_2 are dropped because they represent accesses that occur inside of a static transaction.

1. $(s_1, \delta_{\mathrm{B}[b]}); (s_2, \delta_{id}) = (s_1, \delta_{\mathrm{B}[b]})$
2. $(s_1, \delta_{id}); (s_2, \delta_{\mathrm{E}[e]}) = (s_2, \delta_{\mathrm{E}[e]})$
3. $(s_1, \delta_{\mathrm{E}[e]}); (s_2, \delta_{\mathrm{B}[b]}) = (s_1, \delta_{\mathrm{EB}[(e,b)]}, s_2)$
4. $(s_1, \delta_{\mathrm{B}[b_1]}); (s_2, \delta_{\mathrm{EB}[(e_2,b_2)]}, s_3) = (s_1 \cup s_2 \cup s_3, \delta_{\mathrm{B}[b_2]})$
5. $(s_1, \delta_{\mathrm{EB}[(e_1,b_1)]}, s_2); (s_3, \delta_{\mathrm{E}[e_2]}) = (s_1 \cup s_2 \cup s_3, \delta_{\mathrm{E}[e_1]})$
6. $(s_1, \delta_{\mathrm{EB}[(e_1,b_1)]}, s_2); (s_3, \delta_{\mathrm{EB}[(e_2,b_2)]}, s_4) = (s_1 \cup s_2 \cup s_3, \delta_{\mathrm{EB}[(e_1,b_2]}, s_4)$
7. otherwise $(s_1, \delta_1); (s_2, \delta_2) = (s_1 \cup s_2, \delta_1; \delta_2)$

Using this modified composition of δ-transformers, we perform a second fixpoint calculation. All non-transactional field accesses (i.e., s_{access}) are the annotations associated with δ_{id} and δ_{BE} in the dataflow transformer on the `main` function's exit node.

Finally, we perform the check with respect to s_{xact} and s_{access} described in §7.1 to see if there are potential inconsistencies with respect to the use of static transactions when accessing the fields of a shared record. If there are no inconsistencies, the program is free of data races.

7.3 XProtect Experiments

For the example program in Fig. 2, XProtect emits twelve warnings. Eight of these are false positives. This is due to the fact that the program initializes the `x` and `y` fields of the shared `ColoredShape` record `s`. These false warnings are a common problem when attempting to statically detect race conditions in concurrent programs. Adopting the policy of tools such as Eraser [34], which make a special case for the initialization of global variables, would reduce the number of false positives.

The other four warnings indicate that the `x` fields are accessed in the static transaction (`n1`,`n4`) and also accessed when not in a static transaction. This alerts the programmer of potential data races on the `x` fields of the shared `ColoredShape` record `s`. It is interesting to note that the warnings indirectly alert the programmer that he mistakenly forgot to guard the call to `sumx` with a software transaction.

Again, because we use WPDSs as our dataflow-analysis solver, we take advantage of their witness-tracing capabilities. (A witness trace is a "proof" for a computed dataflow value and encodes the set of program paths that gave rise to it, along with their intermediate dataflow values.) We extract from the witness trace a path that accesses a field of the shared record while not in a static transaction. We present to the programmer this path as well as a static transaction that accesses the same field in the opposite manner (read vs. write).

We note that due to the imprecision of pointer analysis and the abstract model of the program that we analyze (i.e., we do not interpret conditions), a warning does not guarantee that there exists an inconsistency. Even our small test example contained false positives. Alternative techniques, such as software model checking, might be able to find true protection violations.

8. Related Work

The use of tranformers for performing interprocedural-dataflow analysis was introduced by Cousot and Cousot [11] and Sharir and Pnueli [37]. Sagiv et al. showed how this framework, where the transformers are relations annotated with "values" that exhibit certain algebraic properties, can be used to precisely solve instances of copy propagation, namely copy-constant propataion and linear-constant propagation [32]. We used this framework for analyzing programs that adhere to our programming model.

Abstract interpretation was defined in the seminal work by Cousot and Cousot [10]. The analyses that we present can be viewed as an abstract interpretation of the program, where the abstract semantics of a program statement is its effect on the property automaton that checks a policy for correct usage of the transactional primitives.

XOrder analysis is a program optimization whose goal is to reduce false conflicts of a transactional program executing on a hardware ISA that supports transactional memory. The HTM implementation we target performs conflict detection through the cache-coherence protocol [24]. Chilimbi, Davidson, and Larus also used record-layout reordering as an optimization to improve a program's effective use of hardware caches [5]. Their analyses incorporated profile information attained from runs of the program. Our optimization, XOrder, is a purely static analysis. In contrast to the work by Chilimbi et al., we reorder a record type's layout according to its accesses in static transactions. It is left to future work to determine if incorporating dynamic profiling information into XOrder's record layout algorithm could improve our optimization.

Recent work by Harris et al. [18] and Adl-Tabatabai et al. [1] was concerned with optimizing a program's use of software transactional memory (STM). Their work is based on the observation that, by breaking down the inserted calls to the STM's runtime system, standard compiler optimizations, such as dead-code elimination and common subexpression elimination, can be used to optimize a program with respect to the number of calls it makes to the STM's runtime system. XOrder analysis is also concerned with optimizing a program that runs on a system that supports transactional memory. In our case, we optimize the program itself (i.e., the layout of the record types), and not the inserted program statements. It would be interesting to see if our techniques would result in additional performance gains in these other systems.

There have been many tools that detect data races. Eraser by Savage et al is a tool that dynamically detects race conditions [34]. RacerX by Engler et al. uses software model checking techniques to statically find data races in a concurrent program [13]. Most similiar to our analysis, XProtect, is the LOCKSMITH tool by Pratikaks et al. [26]. Their tool infers a consistency correlation on shared objects in a program. A program is data-race free if it all accesses to shared objects are consistently correlated. Whereas they use an extended type system to provide this guarantee, XProtect guarantees data race freedom by by verifying that all accesses to shared records (objects) are consistently protected by a static transaction. This is accomplished by extending XRef analysis with a sound pointer analysis.

References

[1] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI*, 2006.

[2] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA*, 2005.

[3] C. Blundell, E. C. Lewis, and M. M. Martin. Deconstructing transactional semantics: The subtleties of atomicity. In *WDDD*, 2005.

[4] H. Chen and D. Wagner. Mops: an infrastructure for examining security properties of software. In *CCS*, 2002.

[5] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *PLDI*, 1999.

[6] J. Chung, H. Chafi, C. Cao Minh, A. McDonald, B. D. Carlstrom, C. Kozyrakis, , and K. Olukotun. The common case transactional behavior of multithreaded programs. In *HPCA*. 2006.

[7] S. Conchon, J.-C. Filliâtre, and J. Signoles. Le foncteur sonne toujours deux fois. In *Seizièmes Journées Francophones des Langages Applicatifs*. INRIA, 2005.

[8] K. D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *PLDI*, 1988.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[10] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *POPL*, 1977.

[11] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In *IFIP*, 1978.

[12] P. Damron, A. Fedorova, Y. Lev, V. Luchango, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS*, 2006.

[13] D. R. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP*, 2003.

[14] D. R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, 2000.

[15] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, and J. D. Davis. Transactional memory coherence and consistency. In *ISCA*, 2004.

[16] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA*, 2003.

[17] T. Harris, S. Marlow, S. L. P. Jones, and M. Herlihy. Composable memory transactions. In *PPoPP*, 2005.

[18] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *PLDI*, 2006.

[19] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC*, 2003.

[20] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

[21] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *PPoPP*, 2006.

[22] J. Lind-Nielsen. BuDDy - a binary decision diagram package, 1997. http://sourceforge.net/projects/buddy.

[23] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, 2005.

[24] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *HPCA*, 2006.

[25] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of c programs. In *CC*, 2002.

[26] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: Context-sensitive correlation analysis for race detection. In *PLDI*, 2006.

[27] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *ASPLOS*, 2002.

[28] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *ISCA*, 2005.

[29] T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *SCP*, 58, 2005.

[30] T. W. Reps. Undecidability of context-sensitive data-independence analysis. *ACM Trans. Program. Lang. Syst.*, 22(1), 2000.

[31] M. F. Ringenburg and D. Grossman. Atomcaml: first-class atomicity via rollback. In *ICFP*, 2005.

[32] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comp. Sci.*, 167, 1996.

[33] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *PPoPP*, 2006.

[34] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *Theor. Comp. Sci.*, 15(4), 1997.

[35] S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, TUM, 2002.

[36] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *POPL*, 1997.

[37] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, chapter 7. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[38] N. Shavit and D. Touitou. Software transactional memory. In *PODC*, 1995.

[39] B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, 1996.

[40] Wisconsin Multifacet GEMS Simulator. http://www.cs.wisc.edu/gems/.

[41] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *ISCA*, 1995.