# Transacting Pointer-based Accesses in an Object-based Software Transactional Memory System

David Detlefs, Lingli Zhang
Microsoft Corporation
{ddetlefs,lingli.zhang}@microsoft.com

**Abstract**

Software transactional memory (STM) systems for object-oriented languages that allow relocating garbage collection (such as languages for the .NET platform, or Java) must decide how *transactional metadata* (*i.e.,* fine-grained locks) is associated with objects. An obvious approach is to locate this metadata with the object, and move it when the object is moved. This approach becomes complicated if the platform allows interior pointers to heap objects, as .NET does: how do we find the transactional metadata that governs an access via an interior pointer? We present an approach that solves this problem via a combination of static and dynamic information. The most important goal of this solution is that it be *pay-for-play:* that programs that do not use transactions are impacted as little as possible.

## 1 Introduction

The basic idea of a software transactional memory system is that each access to memory is controlled by some corresponding *transactional metadata* (*i.e.*, fine-grained transactional locks). The mapping from data to metadata can be accomplished in various ways; two popular choices are *word-based* and *object-based* [3]. Word-based systems generally create a fixed amount of transactional metadata, and map "words" (which actually have typical sizes closer to those of cache lines) by address to corresponding transactional metadata, typically via a modulus operation, in somewhat the same manner that hardware caches map virtual addresses to cache lines. Word-based systems are popular for STMs designed for "native" language (*e.g.*, C and C++), in which objects do not move; one example is the WSTM system of Harris and Fraser [3]. Object-based STMs transact data at object granularity: a transaction writing to some field of an object conflicts with a concurrent transaction reading or writing to different field of the same object. Thus, transactional metadata is associated with each object. *Managed* languages gain safety and productivity by using garbage collection to reclaim storage. Generally this assumes the ability to move objects during collection to reduce fragmentation. We believe that the use of relocating garbage collection essentially mandates the use of an object-based STM. Associating transactional metadata with object identity, independent of address, allows the mapping from objects to metadata to be unaffected by garbage collection. In contrast, in a word-based system, a relocating collection changes the object-to-metadata mapping (since it changes the object-to-address mapping, and the address-to-metadata mapping stays fixed). So, among other problems, transactions that did not conflict before a collection might conflict after. Having garbage collection potentially abort transactions seems a bad choice – a very popular GC configuration is a generational collection in which some form of relocating collection is used on a frequently-collected youngest generation. Ruling out this GC design choice for transactions would be unfortunate.

Native languages permit full-featured manipulation of, and memory access via, pointers.[1] One of the attractions of word-based STM for native languages is that they have no difficulty with pointer-based accesses: for every access, the code computes the accessed address, and then, if necessary, acquires a lock in the corresponding transactional metadata before performing the access. While some managed languages (*e.g.*, Java) have no pointer-like features, others (*e.g.*, C#, Modula-3) do. Pointers can be created that point into the interior of objects, to local variables in stack frames, or to static variables, and passed as arguments to functions that perform accesses via these pointers. Accesses to these different pointer targets may be transacted differently, which presents a problem for the compilation of the called method. We call this problem the "by-ref problem," since in safe .NET code the major mechanism for introducing pointers is the `ref` argument passing mechanism.

**Contributions.** The contributions of this paper are:

- Definition of the by-ref problem, the tension between object-based STM and the need to support some forms of pointer-based accesses.
- A static analysis that tracks sufficient information about the targets of pointers to allow accesses via those pointers to be transacted, and for callers passing pointer arguments to callees to also pass this target information.
- Optimizations that makes this *pay-for-play,* as defined in section 2.

The rest of this paper is organized as follows. Section 2 describes the by-ref problem in more detail. Section 3 shows that how the use of pointers in unsafe code causes similar issues. Section 4 presents the *pointer target analysis* we use to solve this problem, and section 5 shows how the results of this analysis are used at runtime. Section 6 shows an important optimization that eliminates most of the dynamic

[1] This might even be seen as the *definition* of "native language."

```
void Inc(ref int i) { i = i + 1; }

void Bar(ref int j, SomeObj o, int[] a) {
    int k = 0;
    atomic {
        // Assume "SomeObj" has field m_k
        Inc(ref o.m_k);
        // Static field of "Cls"
        Inc(ref Cls.s_k);
        Inc(ref a[j]); // Array elem
        Inc(ref k);    // Local
        // Pass the ref arg to callee
        Inc(ref j);
    }
}
```

Figure 1: The by-ref problem

cost of this technique. Section 7 analyzes the performance of this system. Section 8 discusses related work, and section 9 concludes.

## 2 The by-ref problem

The C# language allows arguments to be passed by reference. The `ref` keyword indicates such a formal argument, and is also used to indicate taking the address of an expression used as an actual argument for such a formal. Obviously, a reference argument is a form of pointer; we will see that this raises issues for object-based transactional memory. The underlying CLR platform implements reference arguments using a more general mechanism called *managed pointers*. These are pointers that may point into the interior of heap objects, and are tracked as GC pointers: if a managed pointer points into a heap object, it causes that object to survive collection, and any variables containing that pointer value are updated appropriately if the referent object is relocated. A managed pointer may only be created by taking the address of some location – new managed pointer values may not be created via pointer arithmetic. Because mapping interior pointers to object heads can be expensive, managed pointers may only be stored in stack variables, so that this mapping process only occurs during stack scanning.

To see the issues raised by reference arguments, consider the example shown in figure 1, which assumes that C# has been augmented with an `atomic` keyword.

In our CLR STM system, transactional access to different categories of locations are handled in different ways. Transactional metadata takes the form of a *transaction management word (TMW)*. Objects contain a TMW as part of a standard object header.[2] Arrays, like objects, have transactional metadata in their object header, but this may reference a sequence of TMW's, one per "chunk" of the array. Static variables, on the other hand, are not part of an object. Each static variable has a fixed TMW that is allocated lazily, on JIT-compilation of the first transactional code that accesses it. The mapping of static variable to TMW is recorded in a global table. Local variables are accessible only by the currently executing thread, and therefore do not require transactional metadata.[3] The compilation of an atomic block saves the initial value of all necessary local variables and restores them on rollback.

In the example of Figure 1, we see several different kinds of expressions passed by reference to the `Inc` method. If these lvalue expressions had been incremented directly in the atomic block, all of the cases described above would be represented: objects, arrays, statics, and locals. If the increment is accomplished via a call to `Inc`, the program should obey the same transactional semantics. How do we compile `Bar` and `Inc` to make this true?

The last call to `Inc` in the example illustrates that this "byref problem" is not confined to transactional code. Whatever solution we adopt will involve transmitting information from callers to callees about targets of reference arguments. In this last case, the ref pointer argument to `Inc` is an input argument to `Bar`. The use of a transaction within `Bar` is an internal implementation detail, which should not be visible to its callers. So if a caller of `Bar` is going to transmit information about the target of `Bar`'s `j` argument, all callers must do so for all reference arguments, since any such argument might eventually be passed into some method that performs accesses via that reference argument within a transaction. When one attempts to add a new feature to a mainstream product programming platform, especially a feature whose initial audience will be small, it is imperative that the implementation of that feature be *pay-for-play:* users of the feature may pay some performance cost, but programs that do not use the feature should not be impacted. This example shows how a transactional memory concern might impact the normal-case efficiency of the language implementation, so we will have to consider this impact carefully.

One solution to the by-ref problem that would be simple for the compiler would be to uniformly call a runtime routine that determines the required information as a function of the pointer value. For example, this routine could determine whether the pointer points into a thread stack, a static variable area, or into the heap. In the heap case, it could also determine the start of the object into which the pointer points – after all, the garbage collector needs to be able to do this for managed pointers. The problem is that these operations, this last one especially, may be expensive. There is a big difference between determining the containing object of a limited number of interior pointers at GC frequencies, and performing this mapping as a standard part of mutator operation. Further, not all garbage collectors will make this facility easily available to mutator code between collections. The Bartok STM [4] uses these techniques, and section 7 quantifies their costs. We do not explore such dynamic mapping techniques further, but focus rather on compiler-based solutions that we expect to be faster at runtime.

[2] A pre-existing *object header word* is used for multiple mutually-exclusive purposes, including holding a hash code and for per-object monitor locking. If an object uses more than one of these facilities, the header is "inflated:" it becomes a pointer to a larger structure that can hold all the necessary data for all uses. Transactional locking fits into this framework as another such use for the object header.

[3] At least in safe code. This property may be violated by unsafe code, but we will consider that a programming error. In the future, we might extend our programming model to allow the programmer to declare that a local variable will be accessed by multiple threads, and treat such a local variable differently than described here.

## 3 Tractable subset of unsafe code

The CLR (like, *e.g.*, Modula-3, but unlike Java) has a full language that includes *unsafe* features, but also a *safe subset.* Code in the safe subset is guaranteed to preserve type safety, even if it is "incorrect" with respect to the programmer's intentions. Code in the unsafe portion, however, may violate type safety, with arbitrary outcomes. Thus, the type-safety of the system as a whole depends on the correctness of this code. The main way in which type-safety may be violated is via *unmanaged pointers.* These are essentially native pointers, as in C and C++. Arbitrary pointer arithmetic is supported. One can take the address of fields/elements of heap objects/arrays, albeit in conjunction with a construct that "pins" the containing object, so it is not moved by GC while the pointer is in use.

In our project, we were somewhat surprised by the extent to which unsafe code is used within the CLR's Base Class Libary (BCL). Sometimes this is for questionable reasons (*e.g.*, the use of pointers to avoid array bounds checks, which might be better handled as a compiler optimization problem), but in other cases it is necessary. For example, the CLR's `String` class uses a representation that requires an unsafe pointer-based implementation. A CLR `String` is much like an array of Unicode characters, except that it contains not only a field for size of the array, but also a field that gives the length of the string, which may be smaller. A `String` object contains fields for these lengths, then a `m_firstChar` field to indicate the first element of the "inline" character array. Methods that access the character array take the address of this first character, and then access other characters via pointer arithmetic. Representing a `String` in safe code would involve an extra level of indirection, since the `String` would necessarily point to a character array. (This is the representation that Java uses.) Avoiding this extra indirection can be a significant optimization, probably worth the potential safety sacrifice.[4]

The `String` case illustrates that it is not sufficient to transact only code within the safe subset of the CLR, since this would preclude use of too many fundamental types. We must therefore be able to transact accesses via pointers as well as reference arguments. In many cases the use of pointers is sufficiently "well-behaved" that this is possible: if we take the address of a field of an object, we can keep track of the association of that pointer value with that object. In other cases, it will not be possible: if a pointer is stored in a globally accessible location, and read from that location later, we won't have information about the target of a pointer. We will refer to the portion of unsafe code that we can transact as the *tractable subset* of unsafe code.

It is important to emphasize one more point about pointers and unsafe code. Even within the tractable subset, we only promise to transact pointer accesses correctly if they are *correct.* Incorrect use of pointers may compromise the behavior of the system in arbitrary ways, so it should not be surprising that transactions might not work in a program with such errors. Correct pointer use, for purposes of transactional memory, requires that if a pointer is created into a managed object/array (or static or local variable), then any pointer derived from this via pointer arithmetic that is dereferenced must also point into that object/array (or static or local variable). Similarly, if a pointer points into unmanaged memory, then so should any derived pointer that is dereferenced. This correctness condition represents no significant restriction: the CLR makes no guarantees about where objects or variables will be allocated, so once pointer arithmetic creates a pointer outside the original target, there are no guarantees about what it points to.

[4] Obviously, the implementation of `String` is quite well tested!

## 4 Pointer target analysis

The previous sections laid out the problem we need to solve. This section presents the first element of our solution: a static analysis that tracks information, tailored to the needs of STM, about the targets of pointer values. As we shall see, this analysis is a somewhat unusual dataflow analysis, in that we allow it to effect the code generation, in ways that make the analysis simpler.

The analysis associates an abstract value we call a *static target info* with each expression of a pointer type. Here are the different kinds of static target infos, and the information they carry:

- **Top**. Indicates the absence of information about the target of a pointer.
- **ObjOffset**(*offset*). Describes a managed pointer into a non-array heap object. The managed pointer points *offset* bytes into the interior of the object.
- **ArrElem**(*arrRefVN*). Describes a pointer into an element of an array. When we compile code that initializes a managed pointer variable to the address of an array element, *i.e.*, an expression like `&aExp[iExp]`, we modify the code generation to allocate a local variable *arrRefVN*, and store the result of evaluating the array reference `aExp` into this variable. This local is immutable, and is uniquely associated with the expression that creates the managed pointer. We create such variables eagerly, relying on a later optimization pass to remove associated code if the variable is in fact unused.
- **Static**(*fieldHandle*). Describes a pointer to a static field; *fieldHandle* is a pointer to the CLR's internal data structure for the field.
- **Local**. Describes a pointer to a local variable. Details of this static target info are discussed in section 5.3.
- **PtrNull**. Describes the null pointer value.
- **UnmgdPtr**(*objVN*). Describes an unmanaged pointer into a heap object. Much as with the **ArrElem** case, the code generation for the code that takes the address and creates the unmanaged pointer value is modified to allocate local variable *objVN*, and store a reference to the object containing the address into this variable.
- **InArg**(*argNum, offset*). Describes a managed or unmanaged pointer that was an argument to the method being analyzed (at argument position *argNum*), plus the indicated extra offset, whose use is described below.
- **Dyn**(*varNum*). Describes a managed or unmanaged pointer value whose target information is specified dynamically, in a local variable *varNum* allocated just for this purpose. This will be explained in more detail below.

In addition, several other kinds of pointer targets are tracked. In all cases, no transacting is necessary for accesses via these pointers; we distinguish them for ease of debugging, to better track and explain *why* no transacting is necessary:

- **CLRPrivate**. Describes a pointer value that is private to the CLR implementation. These are used to implement the language semantics; their use is not visible at the user level. Therefore, their modifications need not obey transactional semantics. For example, some information about instantiations of generic classes is available by following a pointer path that starts with the method table pointer of some object, and the static target analysis is performed on the CLR JIT's internal abstract syntax tree representation of the program after such code has been inserted.
- **RetBuf**. Methods returning structs of sufficient size allocate the return value in the calling method and pass a reference to this struct to the caller, which initializes its value. Pointers to such "return buffers" never require transacting.
- **Readonly**. Some methods of BCL classes return pointer values that point into memory created by memory mapping a file in readonly mode. We annotate such methods to indicate this property, which allows us to perform accesses via such pointers without transacting, since the readonly nature of the referent is enforced by virtual memory hardware.[5]

The analysis is, apart from the points described above where we change the code being analyzed, a standard data flow analysis. For each basic block of the method being analyzed, we track the abstract values, as defined above, stored in the method's local variables at the start of the basic block (the block's *prestate*). All local variables initially have a "bottom" value. The prestate of the entry block is initialized so that any arguments of pointer types are given appropriate **InArg** abstract values, and the entry block is marked as "changed." The analysis now enters a worklist loop, in which it repeatedly picks a changed block, clears its "changed" flag, analyzes it, and merges the poststate values of local variabls are into successor blocks. If this changes the prestate of a successor block, it is marked as changed. The algorithm terminates when there are no changed blocks.

The descriptions of the abstract static target info values given above explained how these values are determined when a pointer is first created, which covers many of the interesting cases. We will not give a complete description of the rules that determine how blocks are analyzed, but rather a partial description of the remaining interesting cases. We will use the notation *Abs(exp)* to denote the abstract value that the analysis ascribes to the expression *exp*.

- Expression of object reference types are interpreted as managed pointers to the heads of objects:
  $type(exp) <: \texttt{Object} \Rightarrow$
  $Abs(exp) = \mathbf{ObjOffset}(\mathbf{0})$
- The CLR supports `struct` types. These differ from classes in that they may appear "inline" in another type, without a pointer indirection. Thus, a class `C` could have an instance field `m_s` of some struct type S. A method `Foo` could have signature `Foo(ref S)`, and there could be a call to it of the form `Foo(ref c.m_s)`. Within `Foo`, the code could create a managed pointer to a field of `s`, with an expression like `&s.m_f`. This is the closest we can come to "pointer arithmetic" with managed pointers. The rule governing this situation is:
  $Abs(exp) = \mathbf{InArg}(n, offset) \ \wedge \ type(exp) = \texttt{S} \Rightarrow$
  $Abs(\&exp.\texttt{m\_f}) =$
  $\mathbf{InArg}(n, offset + \text{offsetof(m\_f,}S))$
  Similar rules in other situations where we first obtain a managed pointer to a struct, and then obtain a managed pointer to a field of that struct (for example, if the struct is a field of an object).
- As discussed previously, unmanaged pointer values derived from other unmanaged pointers via pointer arithmetic are assumed to have the same target as the original pointer:
  $Abs(exp) = \mathbf{UnmgdPtr}(objVN) \Rightarrow$
  $Abs(exp + n) = \mathbf{UnmgdPtr}(objVN)$
- The analysis treats some methods as intrinsics, assuming it knows their semantics. The type `System.IntPtr` is treated as a pointer type, and expressions of this type have an associated static target info value. If an `IntPtr` is constructed from a pointer expression, it gets the static target info of that expression, and if the `IntPtr` is converted back to a pointer by a coercion operation, the target info of the `IntPtr` flows to the new pointer expression.
- Method invocations may return pointer types. For performance reasons discussed in section 6, we assign **Top** to such return values, unless an annotation indicates that the target information of the return value should be tracked. This mechanism is discussed in section 5.2.

Any incompatibility between two static target info values that are merged at a control flow join point conservatively results in the **Top** analysis value. Later we will discuss situations in which this is limiting, and an extension that will allow more accurate information to be tracked, at some runtime cost.

This analysis associates a static target info with each expression of a pointer type. Thus, at a call site for a method that has (managed or unmanaged) pointer arguments, the static target info for the actual argument expressions is sufficient to allow the called method to transact accesses via these pointers, if necessary. The next section describes how this information is transmitted from the caller to the callee.

## 5 Determining pointer target information at runtime

In this section we describe how pointer target information is transmitted from caller to callee. We first implemented a completely general mechanism, which we describe in this section. Unfortunately, this general mechanism has considerable, and non-pay-for-play, runtime overhead. Section 6 describes an important optimization that eliminates this runtime overhead in most cases.

In the general mechanism, at a call site invoking $\texttt{m}(a_1, ..., a_n)$, where some non-zero $k$ of the $a_i$ are pointer

[5] Note again here that we rely on the (unsafe) use of such pointers to be correct – that pointer arithmetic does not derive a new pointer value from such a pointer that points outside the bounds of the region of readonly memory.

expressions, we alter the code generation to allocate a *target info sequence* variable. As shown in figure 2, this has a header, then a sequence of $k$ *dynamic target info* structures, each derived from the static target info for the corresponding pointer argument, in a manner explained below. Before the call, the generated code initializes the header and each of the argument infos, and pushes this target info sequence onto the head of a thread-local linked list of such sequences. After the call returns, we pop the pushed target info sequence from this list. (If the call throws an exception, the exception-handling mechanism removes all pushed sequences whose pops are skipped by the thrown exception.) Obviously, this mechanism is equivalent to passing the target info sequence as an extra argument.

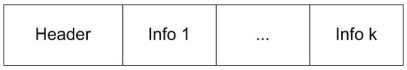


Figure 2: Target Info Sequence

We now explain how static target info values map to dynamic target infos:

- In several cases, dynamic target info structures hold information equivalent to the corresponding static target info. This is true for the **ObjOffset, Static,** and **Top** static target info. It is illegal to do a transacted access via a pointer whose dynamic target info is **Top**.
- The **CLRPrivate**, **RetBuf**, and **Readonly** static target info types translate into a dynamic **None** info – these require no transacting.
- The **Local** static type translates into a corresponding dynamic type, the details of which are discussed in section 5.3.
- The static **ArrElem** and **UnmgdPtr** target infos include names of local variables containing the object into which the described pointer points; these both translate into an **ObjPtr** dynamic target info that carries the actual object pointer value contained in that variable.
- The **InArg** static target info (as well as the **Dyn** info that we will explain below) indicates that the target info for the pointer described is available in a specified location at runtime. For an **InArg**, the pointer value is (or is derived from) a pointer input argument to the current method. The current method therefore has at least one pointer input argument, and its caller must therefore have pushed a target info sequence describing this and any other pointer input arguments. Thus, if an **InArg** is passed to some further callee, when we construct the target info sequence for this call, we copy the information for the input argument provided by the current method's call into the sequence constructed for its callee.

A method that requires access to the target info sequence for its pointer arguments, either to transact an access via that argument, or, as discussed above, to pass an input pointer argument to a callee, calls a helper method that returns the information for the current method. In the scheme described so far (before the optimization described in section 6), this will always be the head info sequence on the linked list associated with the current thread.

### 5.1 Treatment of `this`

Several properties of the .NET environment conspire to require us to pass target info sequences to calls that might not seem to need them, increasing the performance overhead of this scheme. First, instance methods of struct types takes their `this` argument as a managed pointer – since the struct might, for example, be an interior member of another heap object. In a transaction, we might well make accesses via `this` in a struct method; these must be transacted like any other access via a pointer, so we must provide target information for the `this` argument in this case. A struct may be *boxed* into a heap object consisting of a method table pointer followed by the struct. If we invoke one of the struct methods on such a boxed struct, the invocation goes through an *unboxing stub* that adjusts the `this` value to point to the start of the struct, skipping over the method table pointer. Thus, the target information `this` pointer when invoking a struct method on a boxed struct should be an **ObjOffset** indicating an offset equal to the size of the method table pointer (denoted as `sizeof(void*)` below).

A struct may also implement one or more interface types, and struct methods may implement methods of these interfaces. Casting a struct to an interface boxes the struct. A method with an instance of an interface in hand does not know whether the object is a boxed struct whose methods will go through an unboxing stub, and which require target info for their `this` argument, or a normal object that doesn't. It must therefore be conservative and pass the target info information, indicating that the `this` argument, *if* it is a struct `this`, is an **ObjOffset**`(sizeof(void*))`. If the call has other pointer arguments, the target info sequence is just made larger, but, unfortunately, this target info must be passed even if there are no other pointer arguments. This situation is illustrated in figure 3, where the `this` argument in the call to `f.M1` in `M2` is described as **ObjOffset**`(sizeof(void*))`, because the argument *might* be a boxed struct, and the method a struct method, as is in fact the case in the first call to `M2` in `M3`. Because of this case, we must also pass information about the `this` argument in the second call to `M2`, where it will not be used.

In .NET *delegates* are a kind of managed function pointer, and are similar to interfaces with respect to the issues above. Delegates are a limited form of closure: a delegate can be created for an invocation `o.M`, and the delegate instance will retain a reference to the object `o`. If `o` is a boxed struct, an invocation of the delegate will require target info for its `this` argument. As with interfaces, code invoking a delegate instance cannot know whether this is true, so we create target info in such cases, even when there are no other pointer arguments.

Delegates may also be instantiated with static methods, but again, the caller has no idea whether this is the case. Consider a static method with at least one pointer argument. These arguments will appear as pointer arguments in the signature of the delegate type, so the caller will know to pass target info information for them, but will do so in a target info sequence that includes the possibly-required info for the `this` argument. We want the static method to expect the same target info sequence format whether it's called directly or via a delegate. Thus, all target info sequences, even ones

```
interface IFoo {
  void M1();
}

class Foo: IFoo { ... }

struct S {
  int m_i;
  void M1() { ...; m_i++; ... }
}

void M2(IFoo f) {
  ...;
  f.M1();  // this <= ObjOffset(sizeof(void*))
   ...
}

static IFoo g_f;

void M3() {
  S s1;
  g_f = (IFoo)s1; // Boxing operation.
  M2(g_f);
  M2(new Foo());
}
```

Figure 3: Struct methods and interfaces

for static methods that take no `this` arguments, have a slot for a `this` argument.

### 5.2 Tracked pointers, 'dyn' static target Info, and return values

In the discussion of the tractable subset of unsafe code, we stated that if an unmanaged pointer is stored into a globally accessible location, then we cannot assume information about the target of the pointer. In some cases this prevents use of existing code in transactions. To allow such code to be used transactionally with minimal modification, we provide a `TrackedPtr` struct. This type pairs a pointer with the dynamic target info that describes it – a specific kind of "fat pointer."

This type has a `void* Ptr` property, whose `get` and `set` methods are treated specially by the JIT compiler in the following way:

- Calls to `set` method of the `Ptr` property, which takes a `value` pointer argument, are translated into calls to an internal `setValueWithTargetInfo` method, which takes the `value` argument plus an additional dynamic target info argument. The compiler generates code that materializes the dynamic target info corresponding to the static target info that describes the `value` argument, and passes that as the second argument.
- Calls to the `get` method of the `Ptr` property, which returns a pointer value, are translated into calls to an internal `getValueWithTargetInfo` method, which returns the same pointer value, but which also has an additional `out` reference parameter whose type is a dynamic target info struct. A local variable of that type is allocated at the call site, and the address of that variable is passed. The `getValueWithTargetInfo` method initializes this out parameter with the stored target info.

```
int * pi = null;
if (P) {
  pi = &o.m_i;      // ObjOffset
} else {
  pi = &Cls.static; // Static
}
atomic { (*pi)++; }
```

Figure 4: Merge failure

The static target info of the pointer value returned by a `getValueWithTargetInfo` invocation is a new type **Dyn**(*infoVN*). This indicates that the dynamic target info describing the pointer value is contained in the given local variable – the variable allocated to serve as the out parameter in the call. If a pointer value described by a **Dyn** static target info is passed as an argument to a call, the dynamic target info in the specified local variable is copied into the target info sequence – obviously, a call with a **Dyn** actual argument cannot be completely static.[6]

In our experience so far, tracked pointers are most useful in cases where a pointer is stored in a heap location once, and used often, and the pointer has transactable information. For example, in one case, we obtain a pointer into a readonly memory-mapped file, whose contents are interpreted as a (large) struct containing internal arrays. Since this memory is readonly, it does not require transacting. The tracked pointer mechanism allows us to determine this in a principled way, giving some pointers the "readonly" property and using a tracked pointer to store that information with the pointer.

The transformation of single-word set and get operations into operations on multiple words makes atomic operations into non-atomic composite operations. For full correctness, we should synchronize these accesses with some lock, but have not yet done this.

Another situation that would otherwise not provide sufficient data to allow transacting is also solved by the **Dyn** mechanism: pointer return values. We ordinarily treat pointer return results as static **Top** target info values. We allow programmers to add a custom attribute to a pointer-returning method indicating that its return value's target info should be tracked. When this annotation is present, the dynamic target info sequence technique is used, and an extra element is added to the sequence for the return value. The called method fills in this element at all return points, based on its static target info for the pointer return value. During the static analysis, a dynamic target info variable $v$ is allocated to hold the info describing the return value, and the return value is given static target info **Dyn**($v$). The code generated for the call site, copies, after the call, the return value's dynamic target info from its element in the target info sequence into variable $v$.

Finally, we note that the **Dyn** mechanism provides a general mechanism for avoiding "merge failure." Consider the code in figure 4. In the system described so far, this code will be illegal: we will not know how to transact `(*pi)++`, since we won't know whether `pi` is a pointer into an object or a pointer to a static variable – in the static analysis, the control flow merge will assign `pi` the static target info **Top** after the `if` statement. We can avoid this, again, using **Dyn**.

[6] Unless, as discussed previously, we extend the mechanism to allow a dynamic target info to specify an offset in the caller's stack frame.

```
int * pi = null;
DynTargetInfo dtiForPi;
if (P) {
  pi = &o.m_i; // ObjOffset
  dtiForPi = DynTargetInfo(ObjOffset,
                offsetof(typeof(o), m_i));
} else {
  pi = &Cls.static;  // Static
  dtiForPi = DynTargetInfo(static,
                descriptor(Cls.static));
}
// static target info of pi is Dyn(dtiForPi)
atomic { (*pi)++; }
```

Figure 5: Dyn fixes merge failure

When a merge would assign **Top** to a pointer variable that is later used in a way that requires transactable target info, we could transform the code to have accurate information. To illustrate, the code of figure 4 would be transformed to the code of figure 5 We have not yet implemented this transformation.

### 5.3 Local variables

We have postponed detailed discussion of local variables until now. For the most part, local variables are transacted differently than heap objects or static fields. They are accessible only to the current thread,[7] so they do not require transactional locking. They should, however, provide failure atomicity. For local variables updated directly in the body of an atomic block, we save their value before the atomic block, and restore them to their saved values on rollback. (This save/restore mechanism could be restricted to variables live on entry to the atomic block, but we have not yet implemented that optimization.)

Pointers to local variables complicate this story. Our static analysis notes the creation of pointers to locals, and tracks, where possible, the target of such pointers. When a write occurs via such a locally-defined pointer, we add the target local to the set of locals that are saved on transaction entry and restored on rollback, just as if the target local had been written to directly. However, this only partially addresses the issue, because pointer arguments may be defined in caller contexts and passed to callees.

Consider the method `Bar` in figure 6. The static analysis tracks the relationship between `k` and `kp` and saves/restores `k`. (In a program where `kp` might alias multiple locals, the static analysis tracks the set of possible target locals, and all members of the set are saved/restored.)

The situation for `ip` is more complicated. In `Blah`, the variable `i` is initially 7. The first call to `Bar` increments `i`. If it now throws an exception that aborts the transaction, `Blah` should observe `i` restored to its initial value.

We obtain this behavior by describing the argument in `Blah`'s call to `Bar` with a **Local** dynamic target info. When we enter a transaction that contains a possible write via a pointer argument (where passing the pointer as an argument to a called method is considered a possible write), we generate code that dynamically queries the target info for the argument, and if it is a pointer to local, saves the value

[7] At least in safe code – we currently consider code that passes pointers to local variables to other threads a violation of our programming model. We might support local variables used in this way via some declaration in the future.

```
void Bar(int * ip) {
  int k = 100;
  int* kp = &k;
  atomic {
    (*kp)++; (*ip)++;
    // Throw aborts tranaction.
    if (P) throw new Exception();
  }
}

struct S { double m_d; int m_i; float m_f; }

void Blah() {
  int i = 7;
  S s;
  try {
    Bar(&i);
    Bar(&s.m_i);
  } catch (Exception x) {
    int k = i;  // Should be 7, not 8.
  }
}
```

Figure 6: Pointer-to-local issues

of the local, and registers an "undo action" to restore this value on rollback. Since multiple pointer arguments could point to the same local variable, only the first undo action for a given local is registered. One further complication is that the pointer could be into the interior of a local variable of a struct type, as is the case in the second call to `Bar.` The dynamic **Local** target info carries the offset of the pointer into the variable, and the total size of the variable. Thus, at runtime we can determine the address of the start of the local variable, and save/restore the entire local on atomic block entry and rollback, respectively.

## 6 Completely static pointer target information

Allocating, initializing, pushing, and popping target info sequences can impose considerable overhead. We measured a 6X slowdown in one microbenchmark. Again, the caller does not know whether the callee will create a transaction, so, when using the general push/pop technique, we would incur this overhead always, even in programs that do not use transactions. This violates the pay-for-play principle, and would be an unacceptable in a product system where TM might only be used by a few early adopters.

Fortunately, it is usually not necessary to use the general push/pop mechanism. When we compile a call with pointer arguments, before we generated code to push a dynamic target info sequence, we determine whether that target info is actually *completely static:* whether the information that would be put in the dynamic target info sequence is compile-time constant, or requires some dynamic lookup. The static target info types **Top**, **Static**, **Local**, and (importantly) **ObjOffset** translate into completely-static dynamic target infos.[8] The static target info types **UnmgdPtr**, **ArrElem**, and **InArg** do reference information only available at runtime in their translation into dynamic target info types, and thus are not completely static (though we will alter this soon for **InArg**).

[8] This property is the major motivation for representing managed pointers via **ObjOffset**, rather than treating managed and unmanaged pointers uniformly.

If a call's target information is completely static, then we do *not* generated code to allocate, initialize, push, and pop a dynamic target info sequence for that call – the code for the call is the same code generated by the system without transactional memory. Instead, we allocate the target info sequence at compile time, and store it in a global table maintained by the runtime, keyed by the address of the instruction after the call (which the callee will see as its return address). To save space, we also maintain a second table that canonicalizes these target info sequences – since simple ones may describe large numbers of call sites. We now modify the helper method that looks up the target info sequence for a method that requires it. It obtains the return address of its jit-compiled caller, and uses this to attempt to look up its target info sequence in the global table. If this succeeds, it returns a pointer to that sequence; if it fails, we assume that this must indicate that the information at the call site was not completely static, so that the caller used the dynamic linked-list scheme, and return the head of the per-thread linked list of dynamic target info sequences.

In evaluating this mechanism, we found that a signficant fraction of call sites failed to be completely static only because of **InArg** arguments. We therefore modified our system to allow **InArg** target infos to be treated as completely static, creating a dynamic target info of this type. A call with pointer arguments that would have been completely static but for **InArg** values may now be included in the optimization. We only use the dynamic **InArg** target info type when we create a completely static target info sequence to insert into the global table; when we generate code to allocate and initialize a target info sequence, we copy the info for the input argument, as described previously (since the **InArg** information does not tell us *how* to transact via the pointer, only *where to look* for that information). Supporting dynamic **InArg** infos requires a further modification of the lookup algorithm. When the lookup helper finds a static target info sequence for the current return address in the table, it checks before returning (via a flag in the header of the sequence) whether it has any **InArg** elements. If it does, it must resolve those **InArg** descriptors to actionable target infos. It copies the sequence to an allocated copy, and initiates a *stack walk,* in which it retrieves the information for the caller, finds the elements indicated by the current method's target sequence's **InArg** specifications, and replaces those with the indicated elements of the caller's target info sequence. These may, recursively, also be **InArg** infos; the stack walk continues up the call chain until all **InArg** infos have been resolved.

We can imagine techniques for treating the remaining static target info types as completely static. A **UnmgdPtr** static target info names a local variable containing the object into which the unmanaged pointer points. Similarly, an **ArrElem** static target info names a local variable containing the array object. In creating dynamic target infos corresponding to these static target info types, we could convert this variable name into the stack-pointer-relative offset of the variable in the caller's stack frame. The lookup mechanism could find this stack slot. These options require closer integration with the JIT compiler than we have needed so far, and we have not yet found them necessary. Section 7 gives details of the effectiveness of the completely static optimization without this addition.

## 7 Performance analysis

In this section we evaluate the performance of our system. Our main focus is on the pay-for-play impact of this mechanism on normal operation, but we also measure the performance of the mechanism when used within transactions.

### 7.1 Effectiveness of the "completely static" optimization

Table 1 details the effectiveness of the "completely static" optimization. We JIT-compiled all the methods in several managed assemblies, listed in the first column. The second column gives the number of call sites encountered in those methods, and the third and fourth columns the number of those call sites that required pointer target info, and the percentage this represents of all call sites, respectively. The fifth column gives the number of call sites whose pointer target info was completely static, and the sixth expresses this as a percentage of the fourth column. Finally, the seventh column gives the number of sites with length=1, and the eighth the percentage this forms of call sites with pointer target info; this gives a rough indication of the number of call sites that have information only about a `this` argument, for the reasons discussed in section 5.1. There are only a few distinct lengt=1 target info sequences, so canonicalizing these, and other simple cases, likely saves significant space.

Table 1 shows that a fairly consistent fraction, between 12% and 23%, of call sites require pointer target information. A very large majority of these, at least 87%, are completely static, and thus incur no dynamic cost. A fairly large fraction of the call sites, usually about 50%, have pointer target sequences of length 1, so most cases are quite simple.

Another pay-for-play concern is the cost of the pointer target analysis during dynamic compilation. Table 2 shows, for this selection of assemblies, the cost of compilation of all the methods in the assembly, and the fraction of this consumed by pointer target analysis, when pointer target analysis is enabled. This is 11% or less of total compilation time in all cases.

| assembly | comp. time *(sec)* | Anal. time *(sec)* | *pct* |
|---|---|---|---|
| mscorlib | 6.76 | 0.62 | *9.2* |
| System | 5.81 | 0.55 | *9.5* |
| Sytem.Deployment | 1.29 | 0.08 | *6.2* |
| Sytem.Design | 6.96 | 0.67 | *9.6* |
| System.Drawing.Design | 0.44 | 0.04 | *9.1* |
| System.Windows.Forms | 8.37 | 0.94 | *11.2* |
| System.Xml | 5.07 | 0.56 | *11.0* |

Table 2: Pointer target analysis cost

Figure 7 shows the results of several hundred performance regression tests, mostly microbenchmarks, used by the CLR. For each test, we measured the difference between the base non-STM system and the STM-enabled system, for whatever "figure of merit" the test uses (usually time, but sometimes working set or other measures.) To be clear, none of these tests use transactions; these measurements are intended to measure the impact of the *ability* to support transactions on non-transactional code. While it is visually obvious that there is a small general negative trend, and

| assembly | # call sites | # with pointer arg target info | *pct of calls* | # completely static | *pct of calls with info* | info length 1 | *pct of calls with info* |
|---|---|---|---|---|---|---|---|
| mscorlib | 77109 | 12722 | *16.5* | 11877 | *93.4* | 7456 | *58.6* |
| System | 60507 | 8843 | *14.6* | 8535 | *96.5* | 4802 | *54.3* |
| Sytem.Deployment | 11425 | 1995 | *17.5* | 1854 | *92.9* | 1030 | *51.6* |
| Sytem.Design | 87582 | 14945 | *17.1* | 14774 | *98.9* | 4803 | *32.1* |
| System.Drawing.Design | 4827 | 1111 | *23.0* | 970 | *87.3* | 534 | *48.1* |
| System.Windows.Forms | 111890 | 22899 | *20.5* | 22690 | *99.1* | 11250 | *49.1* |
| System.Xml | 56740 | 7039 | *12.4* | 6679 | *94.9* | 5085 | *72.2* |

Table 1: Effectiveness of "completely static" optimization

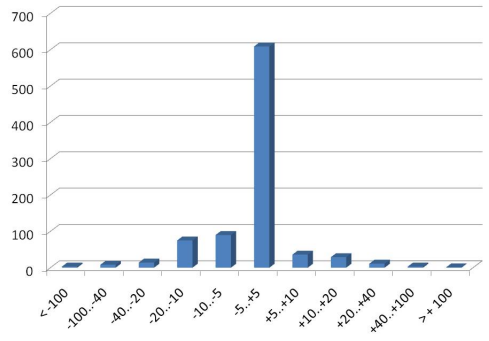


Figure 7: Performance Test Perf Delta Histogram

we have thus not yet completely achieved our goal, we are close: the majority of the tests are within 5% of the original performance. While we have not shown the corresponding comparison without the "completely static" optimization, we can assure you that it was significantly worse.

Finally, we present a measurement of the cost of using pointer target information in transactions. We implemented a dictionary (or mapping) interface, from *key* to *value,* via a binary tree, with two different versions of the lookup operation. One returns a boolean indicating the success or failure of the lookup, and, when true, returns the *value* for the given *key* as an `out` reference parameter. The other requires the *value* type to be an object type, and forbids mappings to `null` so that `null` can indicate a lookup failure. We compared the performance of these dictionary implementations by building trees of $2^{15}$ nodes, then doing a $10^7$ lookup operations of randomly-distributed keys within transactions. In the current system, the dictionary whose lookup operation does not use a reference parameter takes 5.2 secs, while the dictionary whose lookup operation does take a reference parameter takes 8.2 secs when the reference parameter is bound to a local variable in the caller – this is about 58% slower. A major optimization necessary to get this level of performance was the implementation of an intrinsic function to fetch a method's return address. Before this we used a heavyweight general stack-walking mechanism for this, which caused greater than 3X slowdowns. Further optimizations are no doubt possible.

Our colleague Tim Harris was kind enough to help compare our system with the Bartok STM [4] system on the benchmark described above. Since Bartok uses the GC-related technique discussed in section 1, this allows us to compare with that technique. The Bartok system maintains a table with an entry per (4Kb) page, indicating the start of the first object that extends onto that page. This table is updated as part of allocation. To find the object containing an interior pointer, they map the pointer to the containing page, use the table to find the first object on the page, then traverses (contiguously allocated) objects forward until it reaches the object containing the interior pointer. (This mechanism is also commonly used to initiate scanning of objects on dirty cards in generational garbage collectors that use a "card table" to track old-to-young pointers.)

To measure the cost of this technique, we introduced a third version of the dictionary benchmark described above, in which the reference parameter is bound to a field of a heap object. In our system, this increases the benchmark time to 12.7 secs; we believe the cost of determining by-ref information is roughly the same as when the argument is a local variable, but writing to a field heap object turns the read-only transaction into a writing transaction, with corresponding extra expenses. The Bartok system is a different compiler, and the measurements were run on a different machine than described above, so comparisons can only be approximate. That being said, Bartok STM quickly recognizes that a reference argument bound to a local variable is in a thread stack, and requires no transacting; the cost for the non-reference parameter version is nearly the same as the cost when the reference parameter is passed a local variable, between 8 and 9 secs. When the reference parameter is bound to a heap object, however, the cost varies greatly, depending on whether the object is near the start or end of a page. When the object is near the start of the page, the cost increases only to about 10.5 secs, but when the object is near the end, the time can exceed 70 secs. Our system seeks to avoid this variation in performance, which would probably be mysterious to users.

## 8 Related Work

There have been a number of library-based STM implementations for managed languages with relocating garbage collectors, including DSTM [7], OSTM [3], and ASTM [9]. In these approaches, memory accesses are transactional if they obey a prescribed protocol. Pointer-based accesses would ignore that protocol, and are therefore not transacted. SXM [5], and DSTM2 [6] are examples of systems in which STM library calls are inserted implicitly via a transformation phase directed by some annotations. These, too, have a required access protocol, which pointer-based accesses would ignore.

The only other STM system we know of that has sup-

ported object-based locking with relocating garbage collection in a language that allows pointer-based access is the Bartok STM [4]. As discussed above in section 7, Bartok uses GC-related techniques for resolving interior heap pointers, which has performance variability probably unsuitable for operations potentially done frequently in mutator code, such as transactional writes.

The techniques we use are somewhat remiscent of those used by, *e.g.,* CCured [10], Cyclone [8], and SafeC [1]. When static analysis cannot prove the safety of pointer-based accesses, these systems use "fat pointer" representations that carry not only the pointer value but also its legal range; dynamic checks ensure that every pointer access is within this range. Our system associates similar information with pointers – in fact, the information we provide could be used to dyamically check the safety of the pointer accesses to heap objects in unsafe code – but uses this information for a quite different purpose, to transact pointer-based accesses within transactions.

Techniques that have also been used in the compilation of garbage collected languages are also relevant. When compiler optimizations create interior pointers from object pointers, it is useful keep track of the relationship between the two variables, so that the derived pointer can be updated if the base pointer is [2]. This is similar to the present work, where we introduce new variables to hold base pointers when interior pointers are created, and track the relationship between these variables in our static analysis.

## 9 Conclusions

We have pointed out an interesting tension in the implementation of software transactional memory: object-oriented languages with relocating garbage collection argue for associating transactional metadata with objects rather than with addresses, but it is simpler to transact pointer-based accesses in an address-based STM system. The .NET platform has both garbage collection and pointer-based acccess, so we had to resolve this tension when building an STM system for .NET. Our approach was to add extra *pointer target info* data to support the relatively rare pointer-based accesses, and to attempt to make this support as pay-for-play as possible. This entailed static analysis to compute the correct target info for each pointer, and runtime mechanisms for passing this information from callers to callees. We presented a performance evaluation indicating that the "completely static" optimization was effective at removing most STM-related overhead in non-transactional code. Thus, we can support pointer-based transactional accesses with reasonable cost in an object-oriented system with relocating garbage collection.

## 10 Acknowledgements

Our thanks to Tim Harris for speedy cooperation in the Bartok STM measurements described in section 7, and to the anonymous reviewers for pushing to make those measurements.

## References

[1] Austin, T. M., Breach, S. E., and Sohi, G. S. Efficient detection of all pointer and array access errors. *SIGPLAN Not. 29*, 6 (1994), 290–301.

[2] Diwan, A., Moss, E., and Hudson, R. Compiler support for garbage collection in a statically typed language. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation* (New York, NY, USA, 1992), ACM, pp. 273–282.

[3] Fraser, K., and Harris, T. Concurrent programming without locks. *ACM Trans. Comput. Syst. 25*, 2 (2007), 5.

[4] Harris, T., Plesko, M., Shinnar, A., and Tarditi, D. Optimizing memory transactions. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2006), ACM, pp. 14–25.

[5] Herlihy, M. Sxm1.1: Software transactional memory package for c#. http://www.cs.brown.edu/ mph.

[6] Herlihy, M., Luchangco, V., and Moir, M. A flexible framework for implementing software transactional memory. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (New York, NY, USA, 2006), ACM, pp. 253–262.

[7] Herlihy, M., Luchangco, V., Moir, M., and William N. Scherer, I. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing* (New York, NY, USA, 2003), ACM, pp. 92–101.

[8] Jim, T., Morrisett, J. G., Grossman, D., Hicks, M. W., Cheney, J., and Wang, Y. Cyclone: A safe dialect of c. In *ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2002), USENIX Association, pp. 275–288.

[9] Marathe, V. J., Iii, W. N. S., and Scott, M. L. Adaptive software transactional memory. In *In Proc. of the 19th Intl. Symp. on Distributed Computing* (2005), pp. 354–368.

[10] Necula, G. C., McPeak, S., and Weimer, W. CCured: type-safe retrofitting of legacy code. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2002), ACM, pp. 128–139.