Using Ownership to Reason about Inherent Parallelism in Object-Oriented Programs

Andrew Craik and Wayne Kelly

Queensland University of Technology
2 George St GPO Box 2434
Brisbane QLD 4001 Australia
a.craik@qut.edu.au, w.kelly@qut.edu.au

Abstract. With the emergence of multi-cores into the mainstream, there is a growing need for systems to allow programmers and automated systems to reason about data dependencies and inherent parallelism in imperative object-oriented languages. In this paper we exploit the structure of object-oriented programs to abstract computational side-effects. We capture and validate these effects using a static type system. We use these as the basis of sufficient conditions for several different data and task parallelism patterns. We compliment our static type system with a lightweight runtime system to allow for parallelization in the presence of complex data flows. We have a functioning compiler and worked examples to demonstrate the practicality of our solution.

1 Introduction

Imperative programming languages have an inherently sequential semantics, but programs in these languages may contain sections which can be safely executed concurrently. The problem of automatically detecting and exploiting this inherent parallelism is long-standing but still beyond the current state-of-the-art for general programs. The emergence of multi-core computing into the mainstream has only increased the need for solutions. Rapid growth in the number of cores per chip is projected and so scalability of proposed solutions is becoming a key concern. Given the difficulty of the problem, we must find a way to reformulate it so that it becomes more tractable even if we loose some precision. We seek a solution that yields sufficient conditions for parallelism that are permissive enough to be useful while allowing programmers and automated systems to easily reason about inter-procedural data dependencies and inherent parallelism in large complex applications.

Parallelism research has traditionally focused on scientific applications where data-flow analysis has tended to be used to solve complicated array index expressions and pointer may-alias questions. We believe that these traditional approaches have met with limited success outside the realm of scientific applications for two main reasons: (1) the analyses are too fine grained and (2) they do

Funding provided by Microsoft Research and the Queensland State Government

not facilitate abstraction and composition. These traditional approaches employ very complex and detailed dependence analyses which do not support abstraction. This lack of abstraction hinders their ability to reason across method and component barriers. At the heart of the problem is the fact that, traditionally, method signatures provide no information about side-effects. This makes it impossible to reason about inter-procedural dependencies without examining method implementations and all those which may be called. The pervasive use of dynamic linking and late binding in modern componentized software systems further exacerbates this problem.

Current approaches to parallelizing applications tend to follow one of two main schools of thought: (1) statically determine potential conflicts and prevent them from occuring or (2) allow conflicts to occur and incurr a runtime penalty to resolve them. Both approaches have different strengths and weaknesses and have been used to solve different types of problems. In this work we have chosen to use static analysis, but there is also valuable and interesting work in the field of runtime conflict resolution. Ultimately, some combination of these approaches may prove the best compromise.

We address the problem of reasoning about inherent parallelism in the specific context of imperative object-oriented languages for two reasons. Firstly, the emergence of multi-cores means that parallelism will now enter the domain of general purpose desktop and server applications; imperative object-oriented languages dominate this development space. Secondly, the object-oriented programming model provides structure to the memory allocated by the program and we seek to exploit this structure to facilitate reasoning at higher levels of abstraction. The use of these higher levels of abstraction allow our techniques to scale across large and complex applications unlike traditional data flow analysis techniques.

Capturing the side-effects of methods is difficult as they may, directly or indirectly, access a virtually uncountable number of memory locations with no easily describable structure. To simplify reasoning about data dependencies we abstract these effects by exploiting the hierarchical "ownership" relationships which inherently exist within object-oriented programs. Objects contain other objects as part of their representation and we view this as providing a large tree structure to all of the objects in the program's heap. We can, therefore, summarize method side-effects in terms of the subtrees which may be accessed or modified. To reason that two computations are independent (i.e. can be executed in parallel) we can reason about the parts of this ownership tree that could, potentially, be accessed instead of reasoning about the individual memory locations themselves. If the sub-trees accessed are disjoint then there can be no data dependencies. This approach sacrifices some precision so that we can perform inter-procedural dependency analyses in a scalable and composable manner.

We have developed a static type system based on Ownership Types[1–6] which is a type system formulated to capture this "ownership" tree. Our system captures, computes, and validates computation side-effects in terms of these ownerships. Because of the hierarchical nature of ownership, we can describe side-effects at different levels of granularity. The side-effects in turn can be used to statically reason about the presence of inherent parallelism.

Complex inter-procedural data flows in addition to dynamic linking and late binding reduce our ability to statically determine the relationship between some contexts at compile time. Because of this, we have complemented our static type system with a runtime representation of these ownership relationships that allow us to determine the disjointness of effects at runtime in $O(1)$ time. Such runtime tests result in conditional parallelism and allow us to parallelize more cases.

To help demonstrate the efficiency and effectiveness of our system for real applications we have created an extension of the C♯ language with support for ownership and effect annotations (the same could easily be done using Java as the base language). We have implemented a compiler for this extended language that performs type checking and generates parallelized C♯ source code as output. Complete source code for our compiler and runtime system is available from our web site [7] together with some examples that we have applied our system to. Snippets from one of those examples are presented in Section 7 together with runtime results.

The reader is asked to note that this paper only addresses the question of *where* inherent parallelism can be found. We address neither the question of *which* parallelism should be exploited nor *how* best to exploit it.

Our specific contributions in this paper are:

- Application of Ownership Types to the problem of automatically parallelizing programs; the use of which has been suggested, by several authors, but there have been no experiments performed to determine if these reasoning systems work in practice for detecting inherent parallelism [3, 6–8].
- Sufficient conditions for the safe parallelization of data parallel `foreach` loops and several task parallelism patterns based on our framework for abstracting and reasoning about side-effects and data dependencies.
- A lightweight runtime ownership system which allows our techniques to operate in the presence of complex data flows. Our runtime implementation provides effect disjointness tests in constant time.

2 Background

To facilitate discussion of our parallelism analyses in subsequent sections, we first provide the reader with background information on Ownership Types and our static type system. We begin by providing a brief introduction to Ownership Types.

2.1 Introduction to Ownership Types

Consider the following code snippet:

```
private Object[] signers;
public Object[] getSigners() {...return signers;}
```

Note that despite the `private` annotation on the `signers` field, it is possible for the `getSigners` method to return the object referenced by this field. The `private` annotation on the field only protects the name of the field and not the data it contains. This code was the source of the infamous `getSigners` bug in Java 1.1.1 for precisely this reason [9]. Ownership Types [1–6] is one of the systems originally proposed to enforce this kind of protection in a rigorous manner.

Enforcing encapsulation requires each object to track: (1) which object's representation it is part of and (2) which objects are part of its representation. In Ownership Types this tracking is achieved through the notions of *ownership* and *object contexts* (here after referred to as contexts). As Clark, Noble, and Potter eloquently described it, "Each object owns a context, and is owned by a context that it resides within" [1]. This definition creates a tree of ordered contexts rooted in the top context called *world*. Each object has a context in which it may store its representation (the object's *this* context). Encapsulation enforcement in these systems is achieved by only permitting the object itself to name its *this* context. If one is unable to name a context one cannot name the type of a reference to an object in that context.

In the `getSigners` example, the `signers` field would have been denoted as owned by the *this* context which would have prevented its contents being returned and directly accessed by external components. Such invariants are useful from the perspective of parallelism analysis because we can reason that others are not accessing the protected data; that is we have some means of containing the scope of effects.

2.2 Ownership Syntax

Our system's ownership syntax is similar to that used by Effective Ownership Types [6]. Ownership Types are a form of constructed type similar to the idea of generic types. While generic types are constructed by providing a list of actual type parameters, Ownership Types are constructed by providing a list of contexts. Methods are normally parameterized by data values. In generic types, methods can also be parameterized by types; in a similar manner, we allow methods to be parameterized by context parameters. In the case of class definitions, the first formal context parameter in the list, by convention, represents the context that owns the object. Any other formal context parameters, if they exist, can be used as actuals to construct other types used within the class. In our extended C♯ language we support both generics and ownership types, so a class can have both type parameters and context parameters. Our syntax for ownership types uses square brackets for delineating the list of formal context parameters and vertical bars for delineating the list of actual context parameters. Whilst contexts are associated with objects, we cannot refer to the context of arbitrary objects, the only contexts that we can name are the special contexts *this* and *world* and formal context parameters visible in the current lexical scope. Below is an example showing this syntax:

```
class LinkedList<T>[x] {
  private Link<T>|this| head;
  ...
}
class Link<T>[y] {
  private Link<T>|y| next;
  private T dt;
  ...
}
```

In the above example, the head node is part of the representation of the linked-list and so is owned by the *this* context of the linked-list. The next field of the node class is defined recursively to also be owned by the same linked-list object. Simply having a private reference to an object does not imply that you own it. It is up to the programmer to decide the logical ownership relationships.

We also allow an objects' state to be subdivided into a set of named subcontexts to allow effects to be described at a sub-object level of granularity. Aldrich and Chambers were the first to propose such a subdivision of an object as part of their Ownership Domains system[10]. Section 7 demonstrates how these subcontexts are used in practice.

Finally, it is important to note that the built-in value types like int, double, and string as well user defined value types in the form of structs do not have have owners because they cannot be aliased; they are passed and copied by value not by reference like classes.

2.3 Side-Effects

Our system partitions effects into stack and heap effects. Stack effects only appear as part of local data-dependency analysis. Heap effects are captures by listing the contexts read and written. Programmers are required to specify heap read and write effects as lists of contexts on method signatures. The type rules for our language enforce the invariant that if an expression or statement reads some value on the heap then the context that owns the value or one of its ancestors is included in the computed read effect set and similarly for writes. The type rules for our language can be found in our companion technical report [11].

Note that the scope of effects can be described at different levels of abstraction due to the hierarchical nature of contexts. The scope of effects can be thought of as similar to street addresses. We could describe an effect as being limited to a very precise location, for example 5^{th} Avenue, Manhattan. It is also correct, but less precise, to say that the effect is limited to New York City or indeed to the United States. If we were then to observe an effect occurring in Boston we would know that the effect in New York and the effect in Boston could not interfere because they are in different cities. If, however, we were to observe an effect occurring in New York City, we know that the effect could interfere with our effect on 5^{th} Avenue.

Consider our previous linked-list example, the following shows the syntax for declaring effects:

```
class Link<T>[o] {
  private Link<T>|o| next;
  private T data;
  public Link<T>|o| getNext() reads<this> writes<> {
    return next;
  }
  public T getNextData() reads<o> writes<> {
    return next.data;
  }
}
```

In the above example, the `getNext` method reads a field from the current object which is captured as a read of *this*. The `getNextData` method reads the current object, generating a read effect of *this*. It also causes a read of the object referenced by `next` which is owned by *o*. The read effect contains only *o* because *this* is part of *o*'s representation and so a read of *o* includes a read of *this*.

This idea and style of effect annotation has been used before by other authors for different purposes. Geenhouse and Boyland were amongst the first to propose an effect system in terms of ownership style contexts[8]. Clarke and Drossopoulou extended these ideas to show how effects could be used for the purposes of validating program properties[3]. Lu and Potter have also proposed effect systems for reasoning about programs[6]. Other authors then took these effect systems and applied them to the problem of verifying locking protocols/ordering in already parallelized programs. Examples of such systems include the work of Boyapati, Lee, and Rinard[12] and Cunningham et al.[13]. Note that this is a very a different, and we believe less interesting, problem than the problem of automatically detecting the inherent parallelism in a sequential problem.

To support legacy components written without any context or effect annotation we employ two strategies. By default, any object without an owner is assumed to be owned by *world* and any method that does not have declared effects is assumed to read and write *world*. Such code will prevent parallelization but is guaranteed to be safe. In addition, we have invented a syntactic construct that allows programmers to specify ownerships and effects for existing legacy classes. The added ownership and effect information is treated as programmer assertions which are accepted on trust rather than being verified.

2.4 Separating Ownership from Encapsulation

The original ownership type systems[1] were designed to enforce strong encapsulation. They both provide a notation for describing which objects were owned by which other objects and placed strong restrictions on which contexts could be read or written from other contexts. Our proposed use of Ownership Types can work with such restrictions; however, they are not strictly necessary for our purposes. Like many recent Ownership systems, including MOJO [2] and Jo∃[14], we choose to omit such strong encapsulation enforcement to make programming easier; we only need to track reads and writes of the heap, not restrict them.

3 Ownerships and Data Dependencies

The key idea of this paper is that we can use the overlap of the read and write effect sets of sections of code to determine if data dependencies can exist. Data dependencies can be classified as either flow, output, or anti-dependencies. If the write set of one section of code does not overlap with the read set of some other then a flow dependence cannot exist. Similarly for output and anti dependencies.

When considering the overlap of effect sets we consider stack and heap effects separately as they represent disjoint sets of memory locations. Sets of stack effects overlap if they contain the same local variable or parameter names. Determining if sets of heap effects overlap is harder because context parameters with different names do not necessarily represent disjoint subtrees of the ownership tree. One context's relationship to another can be said to be:

– equal (=) they are one and the same
– dominating (<) one context is directly or indirectly owned by that on the other
– disjoint (|) they appear on different branches of the ownership tree

Two context sets $\overline{S1}$ and $\overline{S2}$ overlap if any of the contexts in the two sets overlap:

$$\text{overlaps}\big(\,\overline{S1}, \overline{S2}\,\big) = \exists s \in \overline{S2}\; \exists t \in \overline{S2}\; \neg\big(\, s \# t\,\big)$$

In some cases we can statically determine that all of the relevant contexts do not overlap and so we can safely parallelize the code. Similarly, in other cases we can statically determine that the effect sets are not disjoint and so we will not try to parallelize the code. In the remaining cases we may not be able to statically determine if the relevant effects can overlap, but we can determine this dynamically with our runtime system. As we will describe in Section 4, we can compute the relationship of two arbitrary contexts in constant time. It is important to note that presence of a runtime system does not require modifications to our static type system or our sufficient conditions for parallelism.

The following section discusses how the relationship between contexts can be tested at runtime. Following our discussion of our runtime system, we will formulate sufficient conditions for parallelism based on the data dependency techniques developed in this section.

4 The Runtime Representation

Consider the following code snippet of a method parameterized with two context parameters:

```
public void method[c1,c2](...) reads<c1> writes<c2> { ... }
```

Note that the relationship between `c1` and `c2` is not known until the method is invoked. There may be some calls where `c1|c2` and others where they are not. Producing code for every possible combination of context relationships is not feasible in general and so we need to be able to ask questions about the relationship of contexts at runtime.

Our runtime system compliments the static system by allowing context relationships to be checked at runtime. Further, each individual context relationship test can be done in $O(1)$ time even though the program may have a theoretically unbounded number of memory locations in use.

4.1 Context Testing

What we are trying to do is to find the relationship between two nodes in a tree. There are three different relationships which we may want to test for: equality, domination, and disjointness. What we are trying to do is determine if one context is included in the ownership subtree rooted at a second context. This problem is analogous to trying to determine if one type is a sub-type of a second type in an object-oriented language with single inheritance; this is known as the type-extension problem[15].

We map contexts to objects at runtime; this means that an object's *this* context is represented by the object itself; the distinction between the *this* context and `this` variable is, therefore, removed at runtime. The naive implementation of a runtime system would have each object maintain a single parent pointer to its owner. Context relationship tests could then be performed by chasing pointers in the same way that Wirth performed type-extension tests in Oberon[15]. This solution consumes a constant amount of space per object and provides constant-time object creation overhead, but $O(n)$ time relationship tests where n is the height of the hierarchy.

The runtime testing of context relationships is a potentially frequently executed operation. We have, therefore, chosen to use Cohen's solution to the type-extension problem[16] which uses Dijkstra's views[17]; each object maintains an array of pointers to its ancestors. This solution allows us to perform relation tests in $O(1)$ time at the cost of $O(n)$ creation time and space per object, where n is the height of the hierarchy. Fortunately, the maximal depth of ownership hierarchies tends to be low according to recent studies applying ownership to larger programs [18]. Alternative hybrid approaches, like the use of skip lists [19], could be used to provide implementations with time and space performance between these extremes.

4.2 Static Test Minimization

Even with the efficient runtime system outlined, it is necessary to minimize the number of disjointness tests required. We use two techniques to achieve this:

Static Reasoning. At compile time there are a limited number of context relationships which are statically known for any given class:

– An object's *this* context is dominated by its owning context
– All of the declared subcontexts are dominated by the *this* context
– All subcontexts of an object are disjoint from one another

This information can be used to make some parallelization decisions at compile time without runtime tests.

Context Constraints. We have added syntax to our language which allow programmers to statically constrain the relationship between context parameters on classes or methods, similar to C♯'s constraints on generic type parameters. The programmer can specify the relationship between contexts to be domination (<) or independence (|). The constraints are preserved by the type system during type extension, abstraction, and overriding. The compiler statically enforces these constraints during type checking. The example below shows a class with such constraints; specifically that o is dominated by d and d is independent of context t.

```
class Foo[o,d,t] where o < d where d | t
```

5 Task Parallelism

Imperative programs are composed out of sequence, selection, and repetition constructs. Selection is an inherently sequential operation in the absence of speculative execution so we focus on the parallelization of sequence and repetition constructs.

If a programmer has a set of operations that need to be performed, the imperative paradigm requires them to be listed in some arbitrary sequence thereby imposing a total order on the them. In actuality, the data dependencies between operations may only imply a partial order to the steps. The difference between this partial order and the total order represents potential for parallelism. We can construct the partial order by computing the data dependencies between operations. Our effects system allows us to build a Data Dependency Graph (DDG) easily to allows to detect and exploit this parallelism.

6 Loop Parallelism

Repetition parallelism can take many different forms. In this paper we focus on data parallel loops; those in which the structure of any available parallelism is based around the data. In C♯ data parallel loops most commonly take the form of the `foreach` loop which are the only type of loop considered in this section. We will present sufficient conditions for two parallelism patterns for such loops: (1) *data parallelism* where loop iterations execute independently and are distributed across multiple processors and (2) *pipelining* where the execution of a loop iteration is divided up into stages and distributed across multiple processors.

6.1 Loop Parallelism

The data parallelism pattern can only be safely applied if there are no inter-iteration dependencies. We begin by considering the following simple loop:

```
foreach (T|c| element in collection)
  element.operation();
```

We now state informal conditions which are sufficient to ensure there are no such dependencies:

- **Loop Condition 1:** There are no control dependencies which would prevent loop parallelization.
- **Loop Condition 2:** The objects traversed by the iterator are all different. Note that they all share the same owner so this implies their contexts are all disjoint.
- **Loop Condition 3:** The `operation` only mutates the representation of its "own" element and does not read the state owned by any of the other elements.

Detecting the control dependencies which are the subject of Loop Condition 1 is a much simpler problem than detecting data dependencies; we do not claim any new contribution with respect to detecting control dependencies in this paper. Loop condition 2 can be satisfied in one of two ways. Either we can dynamically test the uniqueness condition just prior to loop execution or we can have the programmer assert the uniqueness condition. In the case of a programmer assertion we have the option of verifying the uniqueness invariant at runtime or turning off such assertion checking in order to improve efficiency. If checked, such a uniqueness invariant could be verified either when an insertion takes place or just prior to when the invariant actually needs to hold. The uniqueness assertion can be made by annotating either the collection itself or its enumerator (a collection may contain duplicates, but if its enumerator only returns unique elements then the condition is still effectively met). The uniqueness annotation could be placed on the collection class, or just on specific instances of that collection class. Which of the above possibilities is used to ensure loop condition 2 is met will depend on programmer preferences and performance considerations - we therefore do not stipulate a single mechanism.

Loop Condition 3 says that the write set of `operation` can contain at most *this*. The read set can contain *this*, but it may also contain other contexts r, provided that we know r to be disjoint from c.

We now more formally state our sufficient conditions for parallelism: Let $\overline{R}$ and $\overline{W}$ represent the read and write effects of `operation`:

1. **Loop Condition 2:** The values in the collection traversed by the iterator are asserted to be `unique` meaning that
   $\forall i \in 1..|\texttt{iterated\_values}|\ \forall j \in 1..|\texttt{iterated\_values}|\ i \neq j \Rightarrow$
   $\texttt{iterated\_values[i]} \neq \texttt{iterated\_values[j]}$
2. **Loop Condition 3:**
   $\forall w \in \overline{W}\ w \leqslant \mathit{this} \wedge \forall r \in \overline{R}\ r \leqslant \mathit{this} \vee \big(r \neq c\big)$

If any one of the conditions is known not to hold, then we must execute the original sequential loop to preserve program correctness. We may not be able to decide if conditions 2 and 3 hold at compile-time depending on the contexts concerned. Context relationships may not be known until runtime and so conditionally parallel code is emitted when this is the case:

```
if (/*runtime test: all r's are disjoint from c*/)
  parallel.foreach (element in collection) { element.op(); }
else
  foreach (element in collection) { element.op(); }
```

Facilitating Upward Data Access. So far we have formulated a sufficient condition for data parallel loops designed to allow reading of disjoint and descendent contexts. We now look at facilitating access to ancestor contexts.

Figure 1 illustrates the ownership tree we would like to be able to support. We have a collection of elements $d_1 ... d_n$ which are owned by some object c. From context c we wish to read data from context r. If context r is not in scope (ie we cannot name it) then we must access r through context b, an upward access.

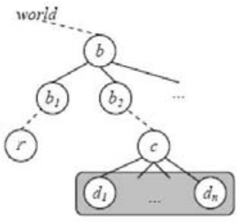

Fig. 1. Ownership relationships between contexts at runtime used for example of capturing context disjointness

Abstracting a safe read of the disjoint context r to be a read of b suddenly makes the read unsafe in our current scheme. To avoid this problem, we introduce the notion of sub-contexts to allow us to partition contexts like b.

With sub-contexts, context b would "own" a finite number of named sub-contexts b_1 and b_2. We only permit the *this* context to be subdivided into sub-contexts. Using these sub-contexts reading r could be summarized as a read of b_1 rather than b itself. If the elements returned by the enumerator are located in sub-context b_2, then we could safely allow the read of b_1 as it is disjoint from c. The idea of sub-contexts has been presented previously by other authors including Clarke and Drossopoulou who used them to provide more precise effect information[3].

Within each class, the programmer can decide if they wish to declare sub-contexts and if they do, they can declare as many as they desire. In the extreme case, each private field might be given its own sub-context, but programmers would more commonly create a sub-context to encapsulate a group of related private fields. The more sub-contexts, the more information that needs to be passed as context arguments on types; the creation of sub-contexts is a trade-off between precision and complexity. Sub-contexts are limited in scope to their class of declaration. To children they look like any other context passed down from the parent while to parents they appear to be part of the owning class' representation.

Loop Body Re-writing. Now that we have explored the sufficient conditions for the parallelization of a simple data parallel loop, the question of how to generalize these conditions to handle arbitrary `foreach` loop bodies arises.

Consider the following loop:

```
class Foo[o] {
  foreach (T|e| elem in collection)
    // sequence of statements possibly including local variable defs
}
```

Fortunately, generalization to arbitrary loop bodies is a natural extension of our existing techniques. We can conceptually re-write the loop body as:

```
class Foo[o] {
  foreach (T|e| elem in collection)
    elem.loopBody|o|(this);
}
```

where `o` is the owner of the class containing the loop and conceptually becomes a method of the element type T:

```
class T {
  void loopBody[c](Foo|c| me) {
    // same sequence of statements replacing all elem by this
    // and all this by me
}}
```

6.2 Pipelining

The data parallelism pattern for loop parallelization can only be applied to loops without inter-iteration dependencies. Consider the following loop:

```
foreach (T|o| elem in collection) {
  S_A; S_B; S_C; S_D;
}
```

This loop may, for example, have both intra- and inter-loop iteration dependencies as depicted in Figure 2. Despite the presence of the dependencies it is possible, for example, to execute iteration 1 of `S_B` in parallel with iteration 2 of `S_A` (provided iteration 1 of `S_A` has already completed execution).

The only form of dependence that we must rule out is a dependence from an iteration p of statement S_i to a later iteration q of some statement S_j where

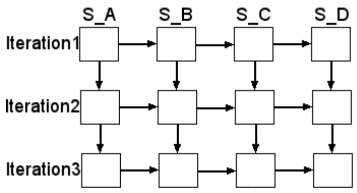

Fig. 2. Diagram showing the permitted data dependencies between stages and iterations. S_A through S_A represent four pipeline stages and $Iteration1$ through $Iteration3$ represent three iterations.

$j < i$. In Figure 2, these dependencies would take the form of diagonal edges moving down and to the left.

To determine which dependencies exist we must first compute the loop body's "virtual" effects of each statement within the loop body using the techniques from Section 6.1. So if a statement reads or writes any part of the representation of the loop iteration variable `elem` then that will show-up as *this* in the virtual effect set of that statement.

We now formalize the sufficient conditions for the safe pipelining of a data parallel loop with stages $S_1..S_n$:

1. the enumerated values are asserted to be unique which means that $\forall i \in 1..|\texttt{iterated\_values}|\ \forall j \in 1..|\texttt{iterated\_values}|\ i \neq j \Rightarrow$ $\texttt{iterated\_values[i]} \neq \texttt{iterated\_values[j]}$
2. There does not exist a dependence (flow, output, or anti) from S_i to S_j where $j < i$. The presence of such dependencies is determined as described previously based on the disjointness of the read and write virtual effect sets of the statements in question.

A number of techniques for detecting and scheduling loops for pipelined execution have been developed over the years [20]. All of these techniques consume a data dependency graph (DDG), like that used for the separation of code blocks into sequences for concurrent execution (see Section 5).

As with full loop parallelization, pipelining relies on specific relationship between the contexts being read and written. If these relationships cannot be statically determined, both the sequential and pipelined versions of the loop can be produced and the choice of which to execute deferred until runtime.

6.3 Data Parallel for Loops

Our techniques only work on data parallel which typically take the form of foreach loops in C♯. We cannot handle arbitrary for loops as they provide no means of associating iterations with distinct data elements. There are, however, some loops expressed as for loops, which are data parallel in nature and could conceptually be converted to foreach. This is not done in many cases due to the semantic restrictions of foreach loops. Specifically, foreach loops only give us access to the value of each element, but do not allow you to change the elements of the collection in place. Further, for loops are often used we need not just the value of each element, but also the index of the element within the collection:

```
for (int i = start; i < list.Count; ++i)
  list[i] = func(i, list[i], ...);
```

To support such cases, we have extended the syntax and semantics of foreach loops, over collections which support indexing, to address both of these problems. The following shows our syntax for expressing such a loop as a foreach loop:

```
foreach (ref ElemType e at Index i in list)
  e = func(i, e, ...);
```

One remaining problem with foreach loops is how to efficiently execute them in parallel across multiple processors. In the case of a for ranging over index values, it is relatively easy to express the subranges to be assigned to each processor. A common approach to parallelizing such loops to use a preliminary inspector phase which sequentially extracts each of the elements prior to the actual loop which then processes partitions of these elements in parallel. In specific cases, the inspector phase can be avoided by using custom collection traversal code. In the case of a list, this produces the same code as the equivalent parallel for loop.

The above syntax can only be used on collections which have specific support for such enhanced iteration. This support can be added to existing classes using C♯'s extension method mechanism and `ref` call parameters (source available on our website[7].

6.4 Proof of Correctness

Finally, in this section we present a proof that loop conditions 1, 2 and 3 as presented in Section 6.1 are sufficient to safely parallelize a `foreach` loop without synchronization. Proofs of the correctness of the sufficient conditions for the other patterns we have presented are very similar and straightforward.

As demonstrated in our technical report [11], our static type system guarantees that if a code fragment directly or indirectly writes a field of an object then the owning context of the object, or one of the contexts which dominates it will appear in the computed write set of the expression. Similarly for read sets.

A loop can be parallelized provided no data or control dependencies exist between iterations. Loop Condition 1 tells us that no problematic control dependencies, such as exceptions, exist. Data dependencies take one of three forms as previously described: output dependencies, flow dependencies, and anti-dependencies.

Assume, by way of contradiction, that an output dependence exists between iterations. The collection must contain two separate elements e_1 and e_2 such that e_1`.operation()` writes to a field of some object x and e_2`.operation()` writes to that same field of x.

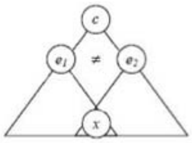

Fig. 3. The relationships between e_1, e_2, and x

The write set of `operation()` may contain only *this*, so we know that e_1`.operation()` can only write to objects that are either e_1 or strictly dominated by e_1. Similarly, e_2`.operation()` can only write object that are either e_2 or strictly dominated by e_2. Figure 3 shows this set of relationships.

Each object is owned by a unique context. If x is dominated by e_1 and e_2, it must be the case that either e_1 dominates e_2 or e_2 dominates e_1. But, e_1 and e_2 are directly owned by the same owner c and $e_1 \neq e_2$ by Loop Condition 1 which provides the contradiction.

Assume now, by way of contradiction, that a flow dependence exists. The collection must contain two elements e_1 and e_2 such that e_1`.operation()` writes to some field x and e_2`.operation()` reads that same field x. We know from the previous step of the proof above that there is no x that is part of both e_1's and e_2's representation.

The only other source of such a flow dependence would be if e_2`.operation()` reads the same field x via some context r such that r is disjoint with respect to e_1's and e_2's owning context c. Figure 4 shows the relationship between c, e_1, e_2, and x.

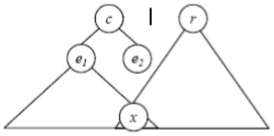

Fig. 4. Relationship of e_1,e_2,c,r, and x and the separation of c and r for the proof of the absence of flow dependencies

So, x is dominated by e_1 which is dominated by c. But x must also be dominated by r which is not possible as $c \# r$. Therefore, no flow dependence can exist. A mirror argument can be made to prove the absence of anti-dependencies.

7 Worked Example

In this section we will present an example of the parallelization of a ray tracing application. This much example demonstrates inter-procedural effect analysis and conditional parallelization based on runtime context relationship testing. The original application was released by Microsoft as part of its Samples for Parallel Programming with the .NET Framework 4 [21]. Note that this example has already been manually parallelized by Microsoft programmers. We are not trying to do a better job of parallelizing the application, we are simply trying to demonstrate that our system can automatically detect the known data parallel loops. Traditional data dependency analysis based systems would struggle to handle the inter-procedural data dependencies found in the program.

The key source of parallelism in this application is the loop in the Render method, where the color of each pixel is determined by tracing rays from the light sources in the scene to the camera. Each ray only reads the state of the scene as its path and color are computed which allows us to trace multiple rays at the same time. The original formulation of this loop is presented below:

```
internal void Render(Scene scene, Color[] scr) {
  Camera camera = scene.Camera;
  for (int y = 0; y < screenHeight; y++) {
    int stride = y * screenWidth;
    for (int x = 0; x < screenWidth; x++) {
      scr[x + stride].color = TraceRay(new Ray(camera.Pos,
        GetPoint(x, y, camera)), scene, 0);
}}}
```

The first problem is that the loop is in not in the form of a foreach loop. Note that the loop is actually iterating over the elements of the `scr` array and so can be transformed using our modified foreach loop syntax. We then add the ownership annotations to this modified loop to produce the following code:

```
internal void Render[s,t](Scene|s| scene, Color[]|t| scr)
  reads<this,s,t> writes<t> {
    Camera|s| camera = scene.Camera;
    foreach(ref Color pixel at int Index in scr) {
      pixel = TraceRay|s|(new Ray(camera.Pos,GetPoint|s|(
        Index % screenWidth, Index / screenWidth, camera)), scene, 0);
}}
```

The power of our system becomes evident when we attempt to determine if the loop can be safely parallelized. In traditional systems the required data dependence analysis would be very complicated because of the aliasing possibilities and number of methods invoked. Our system, on the other hand, allows us to look at the declared effects of the `TraceRay` and `GetPoint` methods. They read contexts `s t` and write nothing. Our compiler and the type system it implements ensures that the method body effects are consistent with the declared effects; the method body cannot cause side-effects not listed in the declared effects. For example, the `Normal` method indirectly called by `TraceRay`:

```
abstract class SceneObject[o] {
  public abstract Vector Normal(Vector pos) reads<this> writes<>;
}
```

```
class Sphere[o] : SceneObject|o| {
 public override Vector Normal(Vector pos) reads<this> writes<>...
}
class Plane[o] : SceneObject|o| {
 public override Vector Normal(Vector pos) reads<this> writes<>...
}
```

Note that our compiler also enforces effect consistency in the presence of overriding so that we do not need to determine which `SceneObject` implementation is being used. The read effect of `this` is abstracted to become context `s` (the owner of the scene) in `TraceRay`.

From the loop body's effects of reading contexts `this`, `s` and `t` and writing `t` and our sufficient conditions, the sufficient conditions for safely executing the foreach loop in parallel are (1) the elements of `scr` are unique, (2) `this` is disjoint from or a child of `t`, and (3) `s` is disjoint from or a child of `t`. Because the relationship between the relevant contexts is not known until runtime, the loop is conditionally parallelized by the compiler subject to these being true.

Table 1. The number of frames rendered per second by the sample ray tracing application. The original sequential and parallel values were obtained from the unmodified application. The enhanced foreach value was obtained from the ownership annotated code.

Implementation	**Frames / sec**
original sequential	0.325
original parallel	0.707
enhanced foreach	0.609

Table 1 shows the average number of frames per second rendered with each of the three implementations. The original parallel and sequential values were obtained using the original code supplied by Microsoft, while the enhanced foreach value was obtained using the automatically parallelized version of the application. The performance runs were conducted on an Intel Core 2 Duo T5800 with 4GB of RAM running Windows 7 Professional 64bit Edition and the .NET 4 Beta 1 runtime. Overall the results show that the performance of the automatically parallelized application is very close to that of the hand parallelized version. There is some performance degradation which can be attributed to a combination of the overheads added by the runtime ownership tracking system and by the overheads incurred in the use of the new enhanced foreach loop; these overheads can be reduced with further development and optimization. Overall, our system is lightweight and efficient at identifying exploitable parallelism.

To annotate the program, after applying the loop transformation discussed earlier in this section, we had to modify 99 lines out of 619 lines of the original application (15%). The majority of these changes were adding method effect lists to method signatures and the addition of context parameters to types. While somewhat burdensome, smart defaulting and ownership inference could reduce this annotation overhead while still providing the benefits outlined previously.

8 Related Work

As was highlighted in the introduction we are not the first to propose capturing effects using ownership contexts nor are we the first to propose many of the language features discussed in this paper. We are the first to apply Ownership Types to the problem of automatically parallelizing existing imperative programs. Others have applied Ownership Types to the simpler, and we feel less interesting, problem of verifying lock ordering in parallelized programs to prevent deadlocks and data races [12, 13]. We now discuss a few of the most directly related works in this area of automated parallelization.

A large body of work has been previously published on the use of local dataflow analysis to extract parallelism from complex iterative algorithms expressed in imperative languages. One example of such work is that of Rus, Pennings and Rauchwerger [22]. These techniques proved very good at extracting fine-grained parallelism from the complex iterative numerical kernels common in the HPC workloads at which they were targeted. Our work has focused on extracting a more course-grained parallelism which scales across method and component boundaries. This kind of course-grained parallelism will become increasingly important as the number of cores per chip grows and more general purpose applications need to exploit parallelism.

Marron, Stefanovic, Kapur, and Hermenegildo proposed techniques for reasoning about data dependencies in Java [23]. Their approach is to perform complex analyses at compile-time on unmodified programs and not try to facilitate programmer reasoning directly as we do. They employ static analysis to determine data dependencies, but need to examine the implementation of any method called to compute dependencies. They do memoize their method analyses, but do not provide the consistency guarantees on overriding like we do and do not make their effects part of the programmers conceptualization. They do, however, handle looping constructs other than `foreach` loops. It is unclear how their techniques would work for large programs.

Various parallel programming languages have also been developed, but they are largely designed for expressing parallelism rather than facilitating the process of parallelizing an existing application by automatically detecting inherent parallelism. X10 [24] is a programming language under development by IBM as part of the DARPA HPCS program which is designed to support parallel programming. Its syntax and features are inspired by Java, but a number of different parallelization and synchronization mechanisms have been purposely included in the language syntax. Our language and X10 serve different purposes. X10 helps programmers familiar with parallelism write and debug parallel applications. We aim to provide a framework for programmers and automated tools to reason about inherent parallelism in sequential programs. Our work and X10 can be viewed as complimentary.

9 Conclusions and Future Work

In this paper we have presented an effects system based on topological ownerships which allow us to reason about the data dependencies in modern imperative object-oriented languages. We have presented sufficient conditions for the safe exploitation of several different patterns of task and data parallelism. We have demonstrated the need for a complimentary runtime ownership system and how such a system can be efficiently implemented.

In the future we hope to expand our techniques to include other looping patterns and continuing to loosen the sufficient conditions for parallelization. To reduce the burden on the programmer we would like to explore automated ownership inference. It would be interesting to explore how our techniques can be applied to C♯'s pointers, possibly similar to Cyclone [25]. We hope to continue to add our annotations to further real applications to gain further understanding in how our extensions affect program development and how much of the available parallelism we are able to successfully exploit. We do not claim that our system is ready for production use, but we feel that some kind of framework to facilitate reasoning about inherent parallelism is necessary. We hope that this work will stimulate further exploration of this space.

References

1. Clarke, D.G., Potter, J.M., Noble, J.: Ownership types for flexible alias protection. In: 13th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 48–64. ACM Press, New York (1998)
2. Cameron, N., Drossopoulou, S., Noble, J., Smith, M.: Multiple ownership. In: 22nd annual ACM SIGPLAN conference on Object-Oriented Programming Systems and Applications, pp. 441–460. ACM Press, New York (2007)
3. Clarke, D., Drossopoulou, S.: Ownership, encapsulation and the disjointness of type and effect. In: OOPSLA 2002: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 292–310. ACM, New York (2002)
4. Potanin, A., Noble, J., Clarke, D., Biddle, R.: Generic ownership for generic java. In: OOPSLA 2006: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, pp. 311–324. ACM, New York (2006)
5. Aldrich, J., Kostadinov, V., Chambers, C.: Alias annotations for program understanding. In: OOPSLA 2002: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 311–330. ACM, New York (2002)
6. Lu, Y., Potter, J.: Protecting representation with effect encapsulation. In: POPL 2006: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 359–371. ACM, New York (2006)
7. Craik, A., Kelly, W.: Mquter parallelism research (2009), `http://www.mquter.qut.edu.au/par`
8. Geenhouse, A., Boyland, J.: An object-oriented effects system. In: Guerraoui, R. (ed.) ECOOP 1999. LNCS, vol. 1628, p. 205. Springer, Heidelberg (1999)

9. Sun Microsystems, Jdk 1.1.1 signing flaw (March 1997)
10. Aldrich, J., Chambers, C.: Ownership domains: Separating aliasing policy from mechanism. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 1–25. Springer, Heidelberg (2004)
11. Craik, A.: Ownership types for reasoning about parallelism - type system and semantics. Technical report, QUT ePrints, Queensland University of Technology (2009), `http://eprints.qut.edu.au/`
12. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: Preventing data races and deadlocks. In: 17th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 211–230. ACM Press, New York (2002)
13. Cunningham, D., Drossopoulou, S., Eisenbach, S.: Universes for race safety. In: 1st International Workshop on Verification and Analysis of Multi-Threaded Java-like Programs (2007)
14. Cameron, N., Drossopoulou, S.: Existential quantification for variant ownership. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 128–142. Springer, Heidelberg (2009)
15. Wirth, N.: Type extensions. ACM Trans. Program. Lang. Syst. 10(2), 204–214 (1988)
16. Cohen, N.H.: Type-extension type test can be performed in constant time. ACM Transactions on Programming Languages and Systems (TOPLAS) 13(4), 626–629 (1991)
17. Dijkstra, E.W.: Recusive programming. Numerische Mathematik 2(1), 312–318 (1960)
18. Abi-Antoun, M., Aldrich, J.: Compile-time views of execution structure based on ownership. In: International Workshop on Aliasing, Confinement, and Ownership in Object-Oriented Programming 2007 (2007)
19. Pugh, W.: Skip lists: A probabilistic alternative to balanced trees. Communications of the ACM 33(6), 668–676 (1990)
20. Allan, V.H., Jones, R.B., Lee, R.M., Allan, S.J.: Software pipelining. ACM Comput. Surv. 27(3), 367–432 (1995)
21. Microsoft Corporation, Samples for parallel programming with the .net framework 4 (May 2009)
22. Rus, S., Pennings, M., Rauchwerger, L.: Sensitivity analysis for automatic parallelization on multi-cores. In: ICS 2007: Proceedings of the 21st annual international conference on Supercomputing, pp. 263–273. ACM, New York (2007)
23. Marron, M., Stefanovic, D., Kapur, D., Hermenegildo, M.: Identification of heap-carried data dependence via explicit store heap models. In: Amaral, J.N. (ed.) LCPC 2008. LNCS, vol. 5335, pp. 94–108. Springer, Heidelberg (2008)
24. Saraswat, V., Nystrom, N.: Report on the experiment language x10. Technical Report 1.7.5, IBM (2009)
25. Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., Cheney, J.: Region-based memory management in cyclone. In: ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, 2002, pp. 282–293. ACM Press, New York (2002)