Obtaining Sequential Efficiency for Concurrent Object-Oriented Languages

John Plevyak, Xingbin Zhang and Andrew A. Chien

Department of Computer Science
1304 W. Springfield Avenue
Urbana, IL 61801
{*jplevyak,zhang,achien*}@*cs.uiuc.edu*

Abstract

Concurrent object-oriented programming (COOP) languages focus the abstraction and encapsulation power of abstract data types on the problem of concurrency control. In particular, pure fine-grained concurrent object-oriented languages (as opposed to hybrid or data parallel) provides the programmer with a simple, uniform, and flexible model while exposing maximum concurrency. While such languages promise to greatly reduce the complexity of large-scale concurrent programming, the popularity of these languages has been hampered by efficiency which is often many *orders of magnitude* less than that of comparable sequential code. We present a sufficient set of techniques which enables the efficiency of fine-grained concurrent object-oriented languages to equal that of traditional sequential languages (like C) when the required data is available. These techniques are empirically validated by the application to a COOP implementation of the Livermore Loops.

1 Introduction

The increasing use of parallel machines has exacerbated the longstanding tension between high-level and low-level programming languages. Though high-level languages ease the task of expressing a computation, advocates of low-level languages argue that detailed control is required to achieve efficiency. Arguably, moving to parallel systems increases both the complexity of programming and the importance of achieving high efficiency. Thus, determining what high level features can be supported efficiently and how to implement them efficiently is an important topic of research.

Concurrent object-oriented programming languages are a promising approach to parallel programming. Recent years have seen the rapid popularization of object-oriented programming techniques for sequential computers, largely because of their benefits in managing program complexity. Concurrent object-oriented programming (COOP) languages focus the abstraction and encapsulation power of abstract data types on managing the complexities of concurrency and distribution. With concurrent objects, which encapsulate their own concurrency control, programmers can safely compose concurrent operations on distributed collections of objects. Unfortunately, to date the best COOP implementations have been inefficient compared to sequential languages.[1]

In this paper, we focus on achieving efficient sequential execution of COOP languages. The larger problem of achieving good parallel performance requires both generation of efficient sequential code and data locality. This latter issue is beyond the scope of this paper.[2] We focus on the former issue, exploring the elimination of object-orientation and concurrency control costs in the generated code. Concurrent object-oriented languages have been inefficient largely because they provide a uniform view of all program data. Even the best implementations incur tens to hundreds of instructions for each method invocation [26, 47] due to the cost of managing a distributed memory (method invocations are location independent) and managing concurrency (locks). Furthermore, the high procedure call frequency typical of object-oriented programs not only magnifies the method invocation overhead, it also reduces the benefits of traditional optimizations.

The overhead of method invocations and concurrency control can be eliminated by aggressive inlining, access region optimizations, and state caching. All of these optimizations are based on excellent (and generally precise) concrete type information [38]. With this set of optimizations, our COOP implementation equals the efficiency of the sequential language C on the Livermore Kernels, a demanding set of numerical benchmarks. While

In the Proceedings of the 22nd Symposium on Principles of Programming Languages (POPL'95), January 1995, San Francisco, California.

[1] We consider only languages that support object-level concurrency. For a discussion of the alternatives see Section 5.

[2] We defer to the wealth of research in this area [30, 39, 22, 36, 3].

the Livermore Kernels do not benefit greatly from object-orientation, all the arrays in the COOP version of the kernels are implemented as concurrent objects, and accessed via object method invocation. Thus to achieve efficiency comparable to C, our compiler must eliminate virtually all of the overhead of concurrency control and object orientation. We believe the performance of our compiler not only exceeds that of all other concurrent object-oriented implementations, but even surpasses many other implementations of sequential object-oriented implementations such as C++.

The specific contributions of this work are:

- Identifying the critical efficiency issues in achieving sequential efficiency in concurrent object-oriented languages,
- A combination and extension of program optimizations which together produce sequentially efficient COOP implementations, and
- A demonstration of these techniques on the Livermore Kernels which provides empirical evidence that COOP languages can be efficient.

The remainder of this paper is organized as follows. Section 2 describes the COOP programming model, execution model, and compiler framework. Section 3 describes a sufficient set of transformations to construct an efficient implementation for COOP programs. In Section 4, we report the results of applying these transformations to the Livermore Loops. Related work is discussed in Section 5, and we summarize the paper in Section 6.

2 Background

We describe the programming model, execution model, and the compiler framework. The mapping of the programming model to the execution model described here is largely conceptual; further information about our approach and actual implementation of COOP can be found in [9, 30].

2.1 Programming Model

The programming model we assume is the synergistic union of Actors [1, 12, 21] and the object-oriented model [17]. Each object can act concurrently to update its own state, create new objects or invoke methods on other objects. An object provides a set of abstract operations, of which only one may be active at a time. This allows objects to control updates to their internal state. Methods (abstract operations) may invoke methods on several other objects concurrently, waiting on the responses only when required by data flow or the programmer. In this way, the programmer can safely and conveniently compose larger parallel abstractions and entire programs. A number of languages share this model [10, 26, 33, 46].

The programming model has three features which contribute fundamentally to its programmability:

- a shared name space,
- dynamic thread creation, and
- object level access control.

A shared namespace allows programmers to separate data layout and functional correctness. Dynamic thread creation allows programmers to express the natural concurrency of the application, leaving the system to map it to the underlying machine. Object-level access control provides a basic mutual exclusion mechanism which can be used to construct larger atomic operations or other synchronization structures. When such *exclusive* methods are invoked on the current object, *self*, they inherit any access privileges the caller might have, enabling recursion in exclusive methods.

2.2 Execution Model

The execution model is based on a set of single-threaded processing elements with local namespaces. Only objects local to a processing element can be accessed directly. The system synthesizes the global namespace of the programming model by detecting and mapping operations on remote objects into communication. The multithreading in the programming model is achieved by multiplexing the processing elements in software. Thus, each processing element can be viewed as a sequential machine augmented with runtime primitives supporting naming, locking, location, and concurrency control. This model matches existing massively parallel processors [42, 13], and we believe it is appropriate for the next generation machines as well.

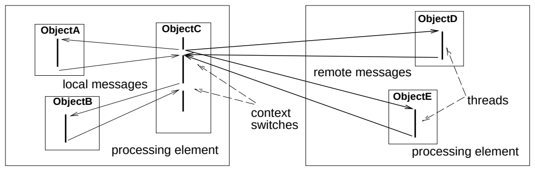

Figure 1: Execution Model Example

Each object has a global name, a lock to implement access control, and a queue for ready and suspended *contexts*. Contexts are heap-allocated activation records which contain a thread's state. When a message is sent, a *future* is created to hold the return value and a thread is started on the target object. When the return value is required, the future is *touched* and the thread suspended until the value is present. Thus, the logical thread within the object may split then rejoin or seem to migrate from processor to processor as in Figure 1.

Basic operations of the execution model such as locking, queuing, and context switching are expensive, but often can be optimized away. For example, a naive approach would create a new thread for each method invocation, but an implementation can execute several threads within their parent to improve efficiency. Thus the concurrency (relaxed serialization) supplied by the programmer can be exploited for parallel speedup, or discarded for sequential efficiency. The runtime exposes the following operations:

- `LOCAL_NAME` converts a global object name to a local name or returns a failure value.
- `TAKE_LOCKS`, given a set of local names, attempts to acquire locks on all the corresponding objects and returns a success or failure value.
- `FREE_LOCKS`, given a set of local names on which locks have been acquired, releases those locks.
- `INVOKE` invokes the specified method, handling all cases (remote objects, locked objects, etc.).

These primitives allow the compiler to test locality and locks inline, opening the door for speculative optimization. They also expose the basic costs in the execution model, enabling many optimizations including some described later in this paper.

2.3 Compiler Framework

The optimizations described in this paper have been implemented as part of the Concert compiler [9]. The intermediate form used in our compiler is the Program Dependence Graph (PDG) [16] in Static Single Assignment (SSA) [15] form. Using the intermediate form, the compiler performs concrete type inference, global constant propagation, cloning, inlining and extension of access regions. Next, instance variables are converted to SSA, and constant folding, common subexpression elimination, and strength reduction are performed. In the back end, the Control Flow Graph (CFG) is reconstructed and the program is translated into Register Transfer Language (RTL). Context slots are allocated and cached in registers, and the RTL is converted into C++, which we use as a portable machine language.

Properties of the intermediate form enable the optimizations described in this paper. Using the PDG, the compiler can determine both the partial order of execution as well as some total order on the contained statements. We say that two access regions (see Section 3.1) are *adjacent* when no other access regions appear between them in the total order. A set of statements are *between* two statements when they are required to execute so by the partial order. The SSA transformation changes variables with storage locations into values. Since our model does not allow arbitrary pointers, only instance variables are associated with storage locations and even these can be converted to SSA within access regions. We say a statement is *functional* when its execution cannot result in the thread blocking, a message being sent, a lock taken, or an update to a storage location.

3 Program Optimizations

In this section, we present three compiler transformations which minimize concurrency overhead for sequential portions of COOP programs. Each optimization exploits information available at compile time to reduce and eliminate runtime overhead. *Inline substitution* of methods eliminates method dispatch overhead and enables intra-procedural optimizations. *Access region expansion* reduces locality and access control overhead. *Context and object state caching* exploit the memory hierarchy of modern microprocessors to reduce multi-threading overhead during sequential execution.

```
for ( l=1 ; l<=loop ; l++ ) {
    for ( k=0 ; k<n ; k++ ) {
        x[k] = y[k+1] - y[k];
    }
}
```

Figure 2: C code for Livermore Loops Kernel 12

Throughout, we use the Livermore Kernels as a benchmark for sequential efficiency. Although the Livermore Kernels do not benefit greatly from object-oriented structure, they are well-known to be a demanding test of a compiler's ability to generate good sequential code. Even a single extra memory reference within the innermost loop can cause a major drop in performance. To illustrate specific optimizations, we use Livermore Kernel 12 shown in Figure 2. The inner loop body contains three array accesses. Because each array is an object in a pure object-oriented language,[3] each array access involves a method invocation. Making these invocations each iteration, particularly in a COOP model, would incur substantial overhead compared to a C implementation. As a running example, we show how this overhead can be removed as a result of the three optimizations.

3.1 Inline Substitution

Inlining is crucial for fine-grained COOP languages because methods are small and general method invocation overhead is high, including procedure call, concurrency control, and even communication overhead. Without inlining, method invocation overhead can easily account for over 95% of a program's

[3]Each array as a whole is an object. Distributed arrays are available through *aggregates* — a concurrent multi-access data abstraction.

execution time. In sequential languages, the main restraint on inlining is the increase in program size. For concurrent object-oriented languages, inlining is constrained by program size, access control, and locality.

A method invocation can be inlined only if the target object is local and can be accessed (any required lock is available). Otherwise message passing or queuing of the message is required. It is not always possible to statically determine these properties. As a result, we speculatively inline method invocations by testing the required properties at run time using the inlining template shown in Figure 3. The template applied to an invocation of method `at` on the object `X` is shown. The runtime primitives `CHECK_LOCAL()` and `TAKE_LOCKS()` check the locality and take the object lock, respectively. Together they define an *access region* under the true arm of the conditional where the object `X` is known to be local and locked. Because the original method invocation is retained in the false arm as a fallback, the inlining template is safe for all method call sites.

```
if( CHECK_LOCAL(X)
   && TAKE_LOCKS(X) )            | runtime guards
   inlined method body of at     | access region of X
   FREE_LOCKS(X)                 |
else
   INVOKE(at, X, i)              | fallback code
```

Figure 3: Inlining Template

When locality or access control information is available at compile time, the inlining template is specialized to eliminate the testing overhead or eliminate unreachable fallback code. For example, no locking is required for immutable objects, and invocations on *self* require additional locking only when the callee is an *exclusive* method but the caller *non-exclusive*. For other objects, the caller need only take the object lock if the target method is exclusive. Similarly, the locality of the target object can be frequently guaranteed at compile-time, as well. Immutable objects and *self* are always local, and the locality of other types of target objects can be estimated using object creation points and the interprocedural call graph.

The inlining template enables inlining at all method call sites where suitable type information for the target object is available.[4] To guide inlining decisions, we use simple heuristics based on static call frequency estimators [44], the size of both the caller and the callee method, and the inline depth. Our experience shows that the simple heuristics combined with compile-time specialization of the inlining template reduces method invocation overhead significantly without excessive code size or compile time. For instance, full optimization of Kernel 12 results in approximately a 50% increase in compile time, 35% decrease in the object code size, and 50% decrease in the the backend C++ compilation time.

[4]We perform global concrete type analysis and customization[38, 6] to bind methods statically in the presence of type-dependent dispatch and inheritance.

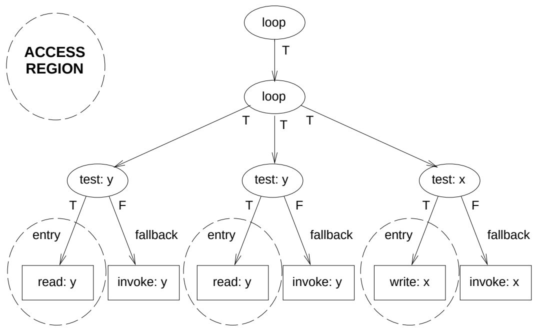

Figure 4: The PDG of Kernel 12 After Inlining

An important result of inlining is the creation of access regions which define a portion of the program in which the two properties, locality and access, are satisfied. Subsequent optimizations build on and leverage off these properties to achieve sequential efficiency. For example, the PDG of the Kernel 12 after inlining (Figure 4) shows an inner loop body consisting of three access regions created by the inlining of the three array accesses. Within each region, a standard suite of sequential optimizations can be applied.

3.2 Expanding Access Regions

Entering the access regions introduced by speculative inlining requires runtime checks which can cost ten or more instructions. If access regions are small or executed frequently the overhead can be severe (as in the loop of Figure 4). In order to reduce this overhead we expand the dynamic extent of access regions. This not only reduces the runtime check overhead but also produces larger basic blocks for classical optimizations. In this section, we consider the general problem of expanding and merging access regions then describe two such optimizations, merging adjacent access regions and lifting access regions above loops and conditionals.

Aspects of the programming and execution model influence these optimizations. Since control flow is structured, the PDG forms a tree of properly nested statements. The access regions are also properly nested, with the locks being acquired and released at the same nesting level. As a result, we can compose access region expansion optimizations from two steps: 1) moving statements into a region and 2) creating an empty region with a particular set of runtime tests. Note that the statements moved in may include conditionals or loop heads directly above the region, expanding the region to include higher levels of the statement nesting. Lastly, execution is non-preemptive with only the runtime context switching, so we need only consider local interactions between runtime primitives.

3.2.1 Correctness

Access-region expanding optimizations must preserve the semantics implied by the original method invocations. This includes preserving the locality and access control properties as well as mutual exclusion of any statements moved into a new region. In addition, we must ensure that neither moving statements nor creating new regions introduces deadlock. These properties are most conveniently discussed within the concurrent systems framework of critical regions [20], monitors [5] and deadlock prevention [23].

Moving Statements into a Region

When moving statements into a region, we differentiate three cases: functional statements (Section 2.3), statements which access storage (non-SSA variables), and potentially blocking runtime primitives. Statements which are functional do not call the runtime nor modify storage so they cannot affect the locality or access properties of a region. Hence, they can be moved safely into any region.

All exclusive storage accesses are conditioned by tests for locality and access control by the programming model. If a storage access is moved into the region the tests for the destination region must subsume the tests for the storage access. Furthermore, if storage accesses for the same object from two distinct regions are moved into the region, they must occur in whole, before or after each other, ensuring locally the mutual exclusion that the programming model guarantees [20]. Together these conditions are sufficient to ensure the exclusion properties of the programming model are preserved.

Potentially blocking runtime primitives cannot be moved into regions unless it can be proven that a resource cycle will not result. This is because blocking operations can give rise to non-local resource deadlock [23]. In the absence of global dependence analysis, correctness can be assured conservatively by preventing such statements from being moved into access regions.

Creating New Regions

Creating an access region containing no statements, but with arbitrary runtime tests, does not change the program providing that no deadlocks are introduced. New deadlocks can only arise if new dependences between locks are introduced. Deadlock can be prevented by obtaining all required locks atomically; that is, all must be availabile for any to be acquired and the entire operation must be executed non-preemptively. Our multi-locking runtime primitive provides this atomicity, avoiding any lock ordering (and thereby avoiding any new lock dependences).[5] Thus, new regions can be created without introducing deadlocks. Subsequently, statements can be moved into the region subject to the constraints above.

[5]Note, that this is not an expensive operation in our model since all objects in multi-locking operation will be local.

The fallback code used when the tests fail must be completely general. For any combination of tests, the fallback code must correctly handle the situation where any of the component tests would have succeeded.

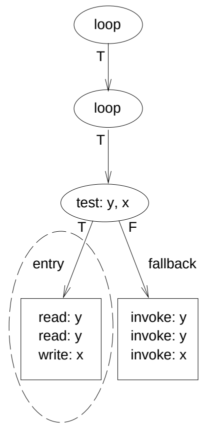

Figure 5: Kernel 12 After Merge

3.2.2 Merging Adjacent Access Regions

Merging adjacent access regions combines the runtime checks for two access regions and merges their code bodies. Merging consists of several steps. First, we create a new region with the combined entrance criteria. Then, using the partial order of execution from the PDG, we identify the code which must execute between the two regions and move it into both the entry and fallback branches of the access test. Finally, we move the entry and fallback code from the original regions into their respective branches of the new region.

The combined entrance criteria represent the conjunction of the checks for the original regions. If the new checks attempt to acquire the lock on a single object twice, the attempt will fail, preserving the mutual exclusion property. However, if at compile time we know two objects are really the same, we can take out a single lock and ensure mutual exclusion by sequencing the operations from the two regions so that they do not interleave. This must-alias determination need only be conservative since the fallback code is completely general.

Note that this approach aggressively merges adjacent regions, so that the optimized path is only be executed when all locks can be acquired at once. Since the cost of blocking on a failure to acquire a lock is large, this optimization extracts high efficiency from the optimized path at a relatively small increase in cost along the unoptimized path. The result of these transformations on the program in

Figure 4 appears in Figure 5. All three of the conditionals have been merged into a single test and two branches, an optimized path and a fallback path.

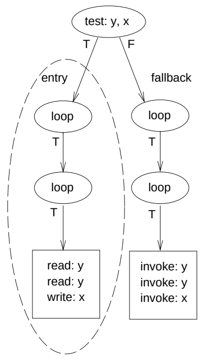

Figure 6: Kernel 12 After Hoist

3.2.3 Lifting Access Regions

Lifting access regions higher in the PDG can improve code efficiency by enabling runtime testing overhead to be removed from loop bodies. In order to ensure correctness, we proceed stepwise as follows. Using a bottom up traversal of the PDG (essentially the program block structure), we attempt to merge adjacent access regions until only one remains within the control dependence region. We then attempt to place the remaining statements within the single access region. If this succeeds we are prepared to lift the regions.

There are only two types of control structures in our intermediate representation: while loops and conditionals. For while loops, the situation is simple. If the control dependence region under the loop is entirely contained in a single access region, the loop header can be moved into the access region. The result is that the access region is lifted over the loop. The same situation holds for single armed conditionals.

```
if( CHECK_LOCAL(x) && CHECK_LOCAL(y)
    && TAKE_LOCKS(x,y) )
    for ( l=1 ; l<=loop ; l++ )
        for ( k=0 ; k<n ; k++ )
            x[k] = y[k+1] - y[k];
    FREE_LOCKS(x,y);
else {
    for ( l=1 ; l<=loop ; l++ )
        for ( k=0 ; k<n ; k++ ) {
            t1 = INVOKE(at, y, k+1);
            t2 = INVOKE(at, y, k);
            INVOKE(putat, x, t1 - t2);
        }
}
```

Figure 7: Example Compiler Output After Lifting on Kernel 12

Conditionals with two arms require that the two regions be merged and lifted simultaneously. The logical steps required to show correctness are: first, break the conditional into two one armed conditionals, one with the negation of the original condition. Then, lift the access regions above these conditionals as above. Next, merge the two resulting access regions. Finally, merge the two one armed conditionals to reconstruct the original conditional.

After inlining and access region expansion, the code within a function or method consists of regions of optimized sequential code. If the program spends the majority of its time in these regions it will be nearly as efficient as a sequential uniprocessor implementation. For example, applying this transformations to our example produces the structure shown in Figure 6. When both **x** and **y** are local, this first loop nest is identical to a sequential program. An example of the code which our compiler might generate appears in Figure 7.

3.3 Caching Object and Context States

Caching both local temporaries (context state) and heap-allocated objects (object state) in registers is required to obtain sequential efficiency. We accomplish this by refining standard register allocation techniques to account for the multithreaded execution model and object level access control.

3.3.1 Caching Context State

Caching context state in a multithreaded execution model is complicated by the possibility of context switching due to synchronization. Because register values are not preserved across context switches, the register allocator must guarantee that when a context switch occurs at a touch, the cached state is saved before the thread yields control. It is also crucial to minimize unnecessary state saving when the thread does not context switch since the amount of register-cached state can be large and touches frequent.

```
          save values in $S - L$ to the context
          TOUCH_BEGIN(Full,future_slot,...)
          save values in $S \cap L$ to the context
          CONTEXT_SWITCH
Restart:  load values in $S \cap L$ into registers
Full:     TOUCH_END
          load values in $L - S$ into registers
```

Figure 8: Lazy State Saving at a Touch

To minimize unnecessary overhead, we save and load register cached state lazily by exploiting the runtime test which determines the context switch. Figure 8 shows our touch template, assuming S and L are the set of values saved and loaded respectively at a context switch. The runtime primitive TOUCH_BEGIN tests the state of the futures. If all futures have values, the code branches to the label Full without blocking; otherwise execution falls through, saving the shared values in $S \cap L$ and yielding control at CONTEXT_SWITCH. When the thread resumes after a context switch, control returns to the label Restart and immediately restores the shared values in $S \cap L$ into registers.

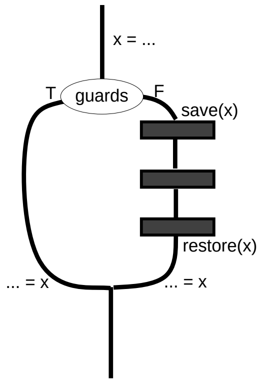

Figure 9: Control flow graph of three access regions merged, with three touches (shaded boxes) in the fallback code.

The possibility of context switching also affects the choice of *live ranges*[11] — throughout which a value is either cached and maintained in a register or kept in memory. Low probabilities of context switching favor live ranges extending over touches; high probabilities favor live ranges delimited by touches, treating touches as function calls in a caller-saved linkage convention. The extensive use of speculative inlining eliminates suspension points inside access regions and increase the likelihood of context switching for suspension points in the fallback code. Therefore, we choose to delimit live ranges by touches and apply a heuristic that caches each live range whose value is accessed at least twice. For example, Figure 9 shows the resulting control flow graph after speculative inlining and merging of access regions at three call sites. Separate live ranges of x delimited by touches allow x to be cached throughout the access region (left) and avoid unnecessary reloading overhead in the fallback code (right).

3.3.2 Caching Object State

We exploit access regions to cache object state in registers safely. Within an access region, the object's state is protected by its lock, preventing accesses by other threads. Thus we can safely cache this state in temporary variables, eliminating memory accesses and requiring only a single update at the end of the access region or before any subsequent method invocation.

For example, Figure 10 shows two possible code sequences for a loop nest traversing a two-dimensional array. The two-dimensional array is constructed from a one-dimensional array with the

```
// Code sequence without object state caching
if( CHECK_LOCAL(a) && TAKE_LOCKS(a) )
  for(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
      ... = a[ a.dimension * i + j];  //a[i][j]
  FREE_LOCKS(a);
else
  ...

// Code sequence with object state caching
if( CHECK_LOCAL(a) && TAKE_LOCKS(a) )
  temp = a.dimension;
  for(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
      ... = a[ temp * i + j];         //a[i][j]
  FREE_LOCKS(a);
else
  ...
```

Figure 10: Comparing two output sequences, one with object state caching (bottom) and one without (top).

instance variable dimension of the object a being used for index linearization. The bottom code uses the properties of access regions to cache dimension in a local temporary temp, potentially saving a memory reference in the innermost loop and enabling other optimizations such as strength reduction. Another advantage of the COOP model is that objects cannot be aliased within the region since an exclusive lock is acquired for each object at run time. In effect, the object level access control serves as a form of non-aliasing declaration, enabling loads and stores to be moved freely within the access region and making the COOP version potentially more efficient than a sequential language version.

4 Results

To demonstrate the effectiveness of these transformation, we compare the performance of our concurrent object-oriented system to a low-level sequential language, C [31]. For the comparison, we use the Livermore Loops, a set of numerical kernels [35] used to measure computation rates for CPU-limited computational structures. All reported numbers are for the third workload of the Livermore kernels at single precision run on a Sparcstation II. The COOP execution times were collected with the UNIX time facility using high iteration counts, and are accurate to within a few percent.

In order to actually test a COOP programming style, we translated the FORTRAN code in a natural object-oriented style. Multi-dimensional arrays were created by subclassing a single dimensional array and using methods to linearize the indexing operations. Since our COOP language does not have pointers, the programmer cannot bypass the encapsulation of the arrays as is typically done in C++ programs to obtain efficiency. We compare

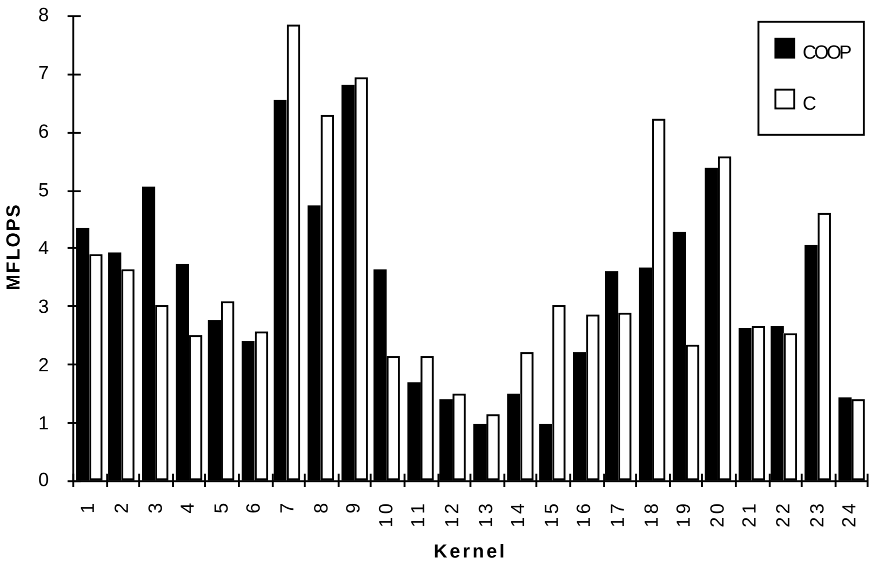

Figure 11: Performance on Livermore Loops

our COOP system's performance against the native C version of the Livermore kernels compiled by the GNU C/C++ compiler.[6] This is the same compiler used by our COOP system as a backend, minimizing differences in low-level optimizations like instruction selection and scheduling.

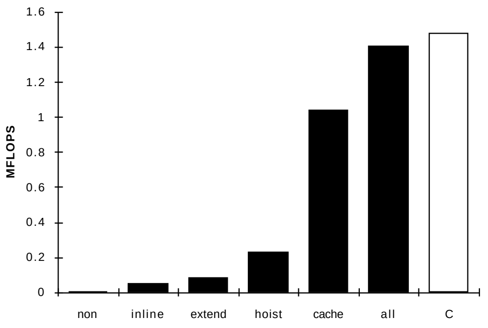

Figure 13: Cumulative Effect of Optimizations on Kernel 12

To illustrate the effect of each optimization we applied each in turn to Kernel 12, and present the performance numbers in Figure 13. Each additional optimization produced a significant increase in performance. With only traditional optimizations, **none**, achieved only several kiloFLOPS. Applying speculative inlining produced an eighteen-fold performance increase, as show by **inline**. Expanding regions by merging increased performance by another 60% while adding lifting access regions brought this to 440%. Caching of context values as in **cache** resulted in 4.5 times performance improvement, coming close to C's performance. The remaining performance gap was traced to our back-end C/C++ compiler being unable in some cases to do common optimizations on the somewhat unnatural code output by our compiler. We implemented these optimizations in our COOP compiler, and the final results **all** include the resulting 40% increase in performance, essentially matching the native C implementation and nearly 500 times better than that achieved by **none**.

Figure 11 contains performance results for all of the Livermore Loops. The performance of the COOP code is quite close to that of the native C code. Essentially all of the object-orientation overhead and concurrency control overhead has been eliminated. Note that this performance exceeds that which would be delivered by most C++ compilers on code written in an object-oriented style. For example, we measured the performance of two representative Livermore kernels in C++ using the GNU C++ compiler. Kernel 12, using virtual functions to access elements in a one-dimensional array, achieves 0.42 MFLOPS — less than a third of the COOP or the C performance. Kernel 21, which operates on two-dimensional arrays achieves 0.32 MFLOPS and even by using non-virtual functions, achieves only 0.45 MFLOPS — less than one fifth of the COOP or C performance.

In Figure 12 we report the performance of the COOP implementations relative to the C implementations ((COOP-C)/C). Of the 24 kernels, our COOP implementation was more than 20% faster on five, the C implementation was more than 20% faster on six, and the remaining thirteen were essentially the same. For the codes where the C compiler gave superior performance, these differences were traced to special purpose array manipulation optimizations in the native C compiler and defi-

[6]We used the highest level of optimization and identical compiler options for all of our measurements.

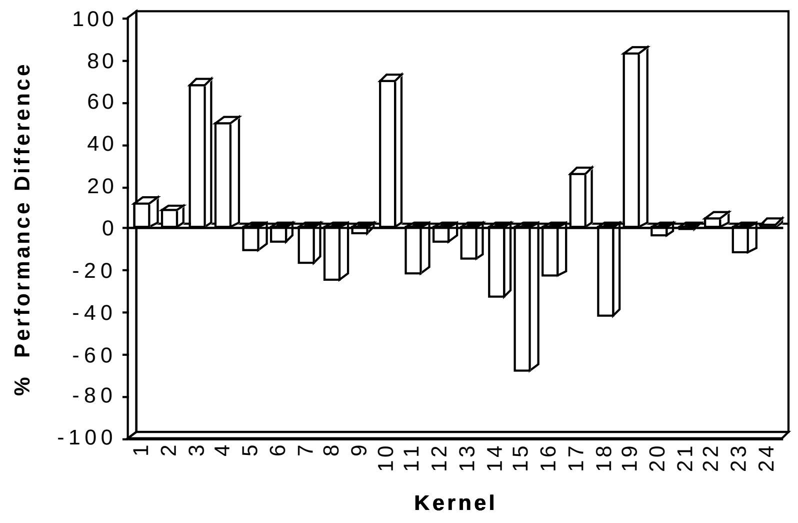

Figure 12: Performance Difference ((COOP-C)/C)

ciencies in our strength reduction optimization (it uses extra registers and does not work for operations under conditionals in loops as in Kernel 15). In cases where the COOP system was faster, the major factor was our ability to apply some optimizations where the C compiler was unable to. The main point of these results is that the COOP model can be essentially as efficient at a sequential C programming model. Any differences that remain are purely in the purview of traditional low level optimization.

5 Related Work

The fine-grained approach to COOP has been studied extensively [33]. In particular, ABCL [45, 46, 47] and Concurrent Smalltalk (CST) [25, 26] were instrumental in helping define the programming and implementation models described here. However, their focus was not on extensive compile-time inter-object transformations. A variety of other parallel object-oriented systems pursue the approach of relying on an underlying sequential language for efficiency [4, 8, 18, 29, 32, 40].

Our work also draws on developments in both the sequential and parallel compiler community. While most of our techniques are familiar ones, we have adapted them significantly to the COOP model. Many researchers have studied inlining for sequential languages [2, 28, 34]; however, their main concern is different from our focus on concurrency and locality. Our inlining techniques are most similar to the ones used in the SELF compiler [6, 7, 24] in their requirement for accurate type information and customization to enable inlining, speculative optimizations, and the insertion of runtime checks to condition optimized code. Our inlining heuristics are a combination of static frequency estimation [44] and the commonly used size constraints. One unique aspect of our inlining transformations is the creation of access regions and the aggressive exploitation of access region properties by subsequent optimizations.

The lifting of access region is conceptually similar to moving loops across procedure boundaries and lifting and blocking of communication in parallel Fortran [19, 22]. In our case, the possibility of deadlock requires atomic primitives and more extensive analysis. Our register allocation scheme is based on that of Chow and Hennessy [11], adapted for lazy state saving. The problem of register allocation in the presence of synchronization points has been studied in dataflow models [14, 41, 43], but the model is slightly different. For instance, TAM has many threads per context, whereas our execution model has only single thread per context, making local analysis around the touches sufficient. The non-aliasing property of an access region's objects inside the region achieves runtime disambiguations of objects. Previous work [27, 37] on runtime disambiguation focuses on memory accesses at the instruction level.

6 Summary and Future Work

We have shown that it is possible to produce efficient implementations from high-level COOP languages, dispelling the myth that such a programming model is inherently inefficient. Using a demanding set of numerical benchmarks, the Livermore Kernels, we have demonstrated that our concurrent object-oriented programming model can achieve good sequential performance. This sequential efficiency forms an important basis for high absolute performance through hardware parallelism. However, it is only half of the solution. The effectiveness of the optimizations depends the data being available (local and not currently in use). Work is underway on both static analyses [39] and runtime techniques [30] to enable the system to ensure availability and thus apply the optimizations in a more informed manner, with the goal of freeing the programmer from the burden of data and task placement.

We have presented a simple programming model and implementation model for a pure concurrent object-oriented language which includes a shared global namespace, dynamic thread creation and object level access control and shown it can be efficient. Our continuing research is directed toward developing additional optimization for array and pointer based data structures through data layout, program analysis and transformation and runtime migration techniques. We are optimistic that through the development of such techniques concurrent object-oriented programming can enable efficient, portable parallel programming.

7 Acknowledgements

We thank Julian Dolby for his help porting the Livermore Kernels and also Vijay Karamcheti and Mahesh Subramaniam for discussions and work on the Concert System. We also thank the reviewers for their valuable comments.

The research described in this paper was supported in part by National Science Foundation grant CCR-9209336, Office of Naval Research grants N00014-92-J-1961 and N00014-93-1-1086, and National Aeronautics and Space Administration grant NAG 1-613. Additional support has been provided by a generous special-purpose grant from the AT&T Foundation.

References

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, 1986.

[2] Randy Allen and Steve Johnson. Compiling C for vectorization, parallelization, and inline expansion. In *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 241–249, June 1988.

[3] J. Bennett, J. B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the Second ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, 1990.

[4] B.N. Bershad, E.D. Lazowska, and H.M. Levy. Presto: A system for object-oriented parallel programming. *Software — Practice and Experience*, 18(8):713–732, August 1988.

[5] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the Association for Computing Machinery*, 17(10):547–557, 1974.

[6] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, pages 146–60, 1989.

[7] C. Chambers and D. Ungar. Iterative type analysis and extended message splitting. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 150–60, 1990.

[8] K. Mani Chandy and Carl Kesselman. Compositional C++: Compositional parallel programming. In *Proceedings of the Fifth Workshop on Compilers and Languages for Parallel Computing*, New Haven, Connecticut, 1992. YALEU/DCS/RR-915, Springer-Verlag Lecture Notes in Computer Science, 1993.

[9] Andrew Chien, Vijay Karamcheti, and John Plevyak. The concert system – compiler and runtime support for efficient fine-grained concurrent object-oriented programs. Technical Report UIUCDCS-R-93-1815, Department of Computer Science, University of Illinois, Urbana, Illinois, June 1993.

[10] Andrew A. Chien, Vijay Karamcheti, John Plevyak, and Xingbin Zhang. Concurrent aggregates language report 2.0. Available via anonymous ftp from cs.uiuc.edu in /pub/csag or from http://www-csag.cs.uiuc.edu/, September 1993.

[11] Frederick C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, October 1990.

[12] William D. Clinger. Foundations of actor semantics. Technical Report AI-TR-633, MIT Artificial Intelligence Laboratory, 1981.

[13] Cray Research, Inc., Eagan, Minnesota 55121. *CRAY T3D Software Overview Technical Note*, 1992.

[14] David Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages an Operating Systems*, pages 164–75, 1991.

[15] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. An efficient method of computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[16] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Laguages and Systems*, 9(3):319–49, July 1987.

[17] Adele Goldberg and David Robertson. *Smalltalk-80: The language and its implementation.* Addison-Wesley, 1985.

[18] A. Grimshaw. Easy-to-use object-oriented parallel processing with Mentat. *IEEE Computer*, 5(26):39–51, May 1993.

[19] Mary W. Hall, Ken Kennedy, and Kathryn S. McKinley. Interprocedural transformations for parallel code generation. In *Proceedings of the 4th Annual Conference on High-Performance Computing*

(Supercomputing '91), pages 424–434, November 1991.

[20] P. Brinch Hansen. Structured multiprogramming. *Communications of the ACM*, 15(7):574–590, July 1972.

[21] C. Hewitt and H. Baker. Actors and continuous functionals. In *Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts*, pages 367–87, August 1977.

[22] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler optimizations for FORTRAN D on MIMD distributed-memory machines. *Communications of the ACM*, August 1992.

[23] Richard C. Holt. Some deadlock properties of computer systems. *ACM Computing Surveys*, 4(3):179–196, Sept 1972.

[24] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 326–336, June 1994.

[25] W. Horwat, A. Chien, and W. Dally. Experience with CST: Programming and implementation. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 101–9. ACM SIGPLAN, ACM Press, 1989.

[26] Waldemar Horwat. Concurrent Smalltalk on the message-driven processor. Master's thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, June 1989.

[27] Andrew S. Huang, Gert Slavenburg, and John Paul Shen. Toward a dataflow/von neumann hybrid architecture. In *Proceedings of the International Symposium on Computer Architecture*, pages 200–210, April 1994.

[28] Wen-mei W. Hwu and Pohua P. Chang. Inline function expansion for compiling C programs. In *Proceedings of the 1989 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246–257, June 1989.

[29] L. V. Kale and Sanjeev Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In *Proceedings of OOPSLA'93*, 1993.

[30] Vijay Karamcheti and Andrew Chien. Concert – efficient runtime support for concurrent object-oriented programming languages on stock hardware. In *Proceedings of Supercomputing'93*, 1993.

[31] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978.

[32] J. Lee and D. Gannon. Object oriented parallel programming. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. IEEE Computer Society Press, 1991.

[33] Henry Lieberman. Concurrent object oriented programming in ACT 1. In Aki Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 9–36. MIT Press, 1987.

[34] Scott McFarling. Procedure merging with instruction caches. In *Proceedings of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 71–79, June 1991.

[35] F. H. McMahon. The Livermore Fortran kernels: a computer test of the numerical performance range. Technical report UCRL-53745, Lawerence Livermore National Laboratory, Livermore, California, 1986.

[36] Stephan Murer, Jerome A. Feldman, Chu-Cheow Lim, and Martina-Maria Seidel. pSather: Layered extensions to an object-oriented language for efficient parallel computation. Technical Report TR-93-028, International Computer Science Institute, Berkeley, Calif., December 1993. (2nd revised edition).

[37] A. Nicolau. Run-time disambiguation: Coping with statically unpredictable dependencies. *IEEE Transactions on Computers*, 38(5):663–678, May 1989.

[38] John Plevyak and Andrew A. Chien. Precise concrete type inference of object-oriented programs. In *Proceedings of OOPSLA*, 1994.

[39] John Plevyak, Vijay Karamcheti, and Andrew Chien. Analysis of dynamic structures for efficient parallel execution. In *Proceedings of the Sixth Workshop for Languages and Compilers for Parallel Machines*, August 1993.

[40] R. J. Smith, II. Experimental systems kit final project report. Technical Report ACT-ESP-077-91, Microelectronics and Computer Technology Corporation (MCC), Austin, Texas., 1991.

[41] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An architecture of a dataflow single chip processor. In *International Symposium on Computer Architecture*, 1989.

[42] Thinking Machines Corporation, 245 First Street, Cambridge, MA 02154-1264. *The Connection Machine CM-5 Technical Summary*, October 1991.

[43] K. Traub. *Implementation of Non-Strict Functional Languages*. Research Monographs in Parallel and Distributed Computing. MIT Press, 1991.

[44] Tim A. Wagner, Vance Maverick, Susan L. Graham, and Michael A. Harrison. Accurate static estimators for program optimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, Orlando, Florida USA, June 1994.

[45] Y. Yokote and M. Tokoro. Concurrent programming in ConcurrentSmalltalk. In Aki Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 129–158. MIT Press, 1987.

[46] Akinori Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System*. MIT Press, 1990. ISBN 0-262-24029-7.

[47] Akinori Yonezawa, Satoshi Matsuoka, Masahiro Yasugi, and Kenjiro Taura. Implementing concurrent object-oriented languages on multicomputers. *IEEE Parallel and Distributed Technology*, pages 49–61, May 1993.