Techniques for Efficient Execution of Fine-Grained Concurrent Programs

Andrew A. Chien, Wuchun Feng,
Vijay Karamcheti and John Plevyak
{*achien,feng,vijayk,jplevyak*}*@cs.uiuc.edu*
Department of Computer Science
University of Illinois at Urbana-Champaign

Abstract
Concurrent object-oriented programming languages are an attractive approach for programming massively-parallel machines. However, exploiting object-level concurrency is problematic as the linkage and communication overhead can overwhelm the benefits of the fine-grained concurrency. Our approach achieves efficient execution by tuning the grain size, matching the execution grain size to that efficiently supportable by the architecture. To verify the feasibility of grain-size tuning, we study the invocation locality of a collection of object-oriented programs. The results suggest that local constraints on placement combined with code specialization can produce a significant increase in execution grain size. We describe several compile-time analyses which identify opportunities to increase grain size. These analyses identify static relationships between objects and enable transformations to reduce invocation cost. Some initial measurements are presented.

1 Introduction

Concurrent object-oriented languages based on the Actor model [2] have received a great deal of attention as an approach for scalable programming of massively-parallel machines because concurrency control and modularity are naturally and conveniently captured in objects.

Two critical implementation inefficiencies have prevented concurrent object-oriented languages from realizing their potential. First, the fine-grained object-level concurrency

The research described in this paper was supported in part by National Science Foundation grant CCR-9209336, Office of Naval Research grant N00014-92-J-1961, 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.

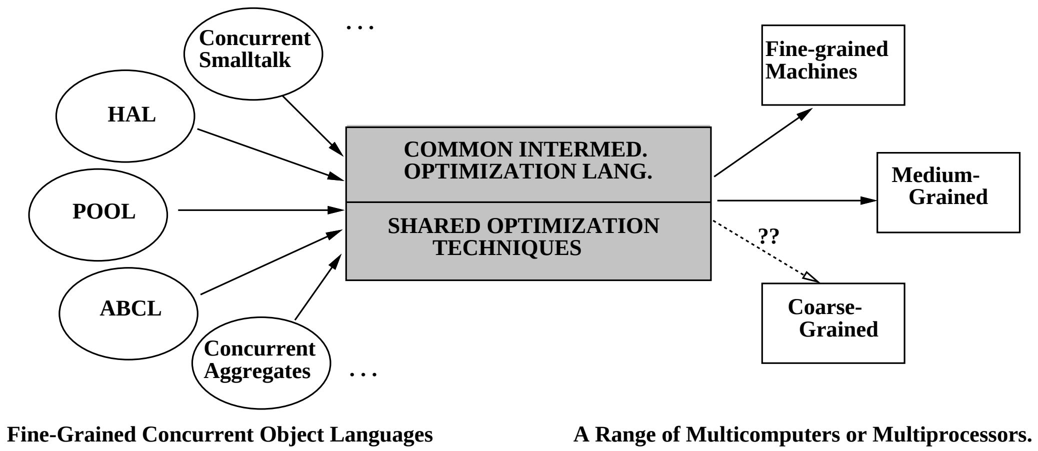


Figure 1: A retargetable, multilingual optimization system for Actor-based languages.

specified in these languages is difficult to implement efficiently, even with hardware support. Second, while allowing objects to be placed and migrated freely gives maximal flexibility for load balancing, oblivious mapping schemes scatter the objects over the system, producing excessive communication and linkage overhead. The ad hoc techniques which are commonly used to address these problems force programmers to explicitly specify granularity and placement of the objects.[1] Though explicit specifications improve execution efficiency on a particular machine, they do so at the cost of portability.

We are developing techniques to analyze and optimize object invocation relations, automatically tuning the execution grain-size to achieve efficient execution. A system based on these techniques would allow programs to be expressed with maximal fine-grained concurrency, thereby providing maximum scalability and portability. Grain-size tuning is a critical element of portability, as it improves processor utilization by reducing the overheads due to scheduling, communication, synchronization, context switching and procedure invocation. A high-level view of the system we are developing is shown in Figure 1.

The cost of an invocation in concurrent object-oriented programs executing on distributed memory machines is typically quite high due to the overheads of message-passing, scheduling, context-switching, type-dependent dispatch, procedure linkage, and the movement of data required for an invocation. In fine-grained object-oriented languages, this overhead can easily dominate the overall cost of the computation. Our approach focuses on developing program analysis methods which identify data locality and transformation

[1] Emerald [5], Rosette [16], and Concurrent C++[15] all distinguish between active (first-class, mobile) and passive(private to an active object) objects.

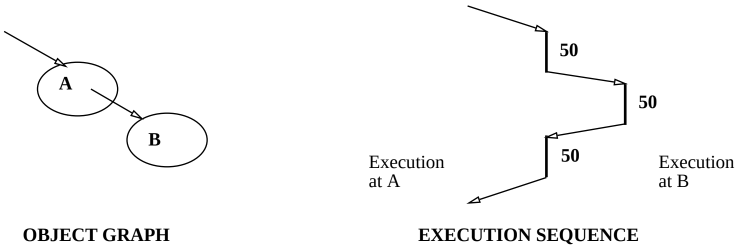


Figure 2: Naive implementation of a call-return sequence between **A** and **B**.

techniques which co-locate or merge objects to take advantage of this locality.

An Example is given in Figure 2. A naive implementation which places objects **A** and **B** on separate nodes would produce an average grain size of fifty instructions. However, the average grain size could be increased to 150 instructions (less the now unnecessary message passing overhead) by co-locating **A** and **B** on the same node. This optimization can be done statically or dynamically and can impact both the execution grain size and program concurrency.[2]

Rather than require programmers to explicitly annotate which objects should be local, our static analysis and run-time techniques deduce this information which is then expressed as a run-time constraint on object placement. Placement constraints allow the compiler to use cheap, local invocation and access mechanisms, improving execution efficiency of the overall computation. In the best case, objects which are wholly contained in another object can be absorbed, allowing the invocation to be replaced with inlined code. Our grain-size tuning techniques optimize control-flow and data locality simultaneously, grouping fine-grained concurrency into larger grains containing only local operations.

If our approach is to be effective, applications must exhibit locality in invocations (i.e., a pair of objects should be involved in a series of invocations). Invocation locality provides leverage, allowing a few transformations to affect a large number of invocations. Co-locating or merging two objects reduces the overhead for all the invocations between the pair. Depending on when invocation locality in a program can be identified, either static (at compile-time) or dynamic (at run-time) techniques can be used. Object migration and dynamic compilation are necessary to exploit the dynamic locality.

In this paper, we have presented the framework for a grain-size tuning system for concurrent object-oriented languages. In Section 2, we describe quantitative studies of

[2]The co-location is an additional constraint on execution concurrency as the processing capacity at each node is finite.

Invocation targets from an object:

B C D E F G H Z Y X Y A M N O P ... (no locality)
B B E F G H C C E F G H F H C C ... (with locality)

Figure 3: Traces of invocation targets from a particular object.

invocation locality in concurrent object-oriented programs. Section 3 describes a variety of static analysis techniques for identifying object relationships at compile-time. The status of a prototype implementation of the grain-size tuning system is detailed in Section 4. Finally, Sections 5 and 6 describe some related work and summarize the paper.

2 Quantitative Studies of Invocation Locality

We define *invocation locality* as the tendency for tasks active at an object **A** to invoke tasks at objects on which they have recently invoked tasks. The objects on which invocations take place due to tasks of object **A** are referred to as the *neighbors* of object **A**. In essence, invocation locality is temporal locality in the targets of invocations from a particular object. For example, the invocation traces shown in Figure 3 illustrate invocation traces with and without locality.

Invocation locality is critical for reducing invocation overhead. With locality, it is possible to specialize common invocation sequences, reducing overhead. There are a number of reasons to expect invocation locality: static hierarchies, data locality, fixed or slowly varying interconnections, and at worst, locality arising from the limited quantity of state an object can address. This section details the results of our quantitative studies of invocation locality, characterizing the locality present in typical object-oriented applications.

The studies cover a collection of concurrent object-oriented programming systems and programs.[3] The invocation patterns in the programs were traced by recording the invoking object, the invocation target, and the message type involved, for each invocation-return sequence. The traces were first analyzed to obtain a per-object view of the locality. Invocations originating from the object are categorized according to the invocation target, and the neighbors of the object are ordered on the basis of the cumulative number of invocations, identifying the most preferred neighbors (frequently communicated with). Summing the invocations to the preferred neighbors for each object, and normalizing by the num-

[3] We have examined CST [12], CA [8], ABCL [18], POOL [3], Rosette [16], and Presto [4] programs.

ber of invocations, yields the overall invocation locality over the program. This aggregate measure of communication to preferred neighbors approximates the reduction in communication which can be obtained by specializing the invocation sequence between an object and its preferred neighbors.

The results of the invocation locality studies are presented in Figure 4. This data is from a wide range of concurrent object-oriented programs written by a variety of programmers. Results of invocation locality for programs in the other systems are not presented due to either a lack of available large programs (POOL, Rosette). For each application, the corresponding row in the table lists the total number of messages in the application, and the fraction of this number that is observed in communications to preferred neighboring objects.

We observe from Figure 4 that significant invocation locality is present in all the applications. For example, the statistics for the ABCL program, **nbody_I** (a tree-code for the n-body interaction problem), show that about half the messages in the system can be eliminated if it were possible to co-locate an object with its first two preferred neighbors. Co-locating objects has the effect of eliminating an remote message send which is the dominant cost in any invocation sequence. From the Figure, it can be seen that a reduction of $20 - 40\%$ in the number of messages is reliably possible for all the applications, just by co-locating an object with its most preferred neighbor.[4] This improvement can be as high as 70% and is typically in the range of 50% of the total number of messages.

The number of messages to preferred neighbors can be translated into an equivalent reduction in invocation overhead under the assumption of optimal pairings of objects; however, such pairings may not be realizable. However, given that the above statistics are for a statically determined preferred neighbor of an object (and not the set of preferred neighbors over an object's lifetime), we are optimistic that this observed invocation locality can be translated into significant reductions in invocation and message-passing overhead for object-oriented programming systems.

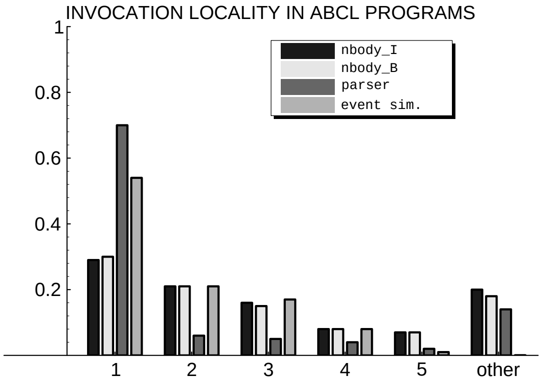
3 Static Analysis

Our analysis techniques must identify safe invocation relations to optimize since grain-size transformations must preserve program semantics. Due to the known difficulty of data-flow and aliasing analysis in the presence of pointers [17, 13], the static analysis techniques

[4]This number is conservative for CA programs in general because the use of aggregates and their randomized interface dissipates invocation locality. Locality can be enhanced by reducing the randomness of the interface.

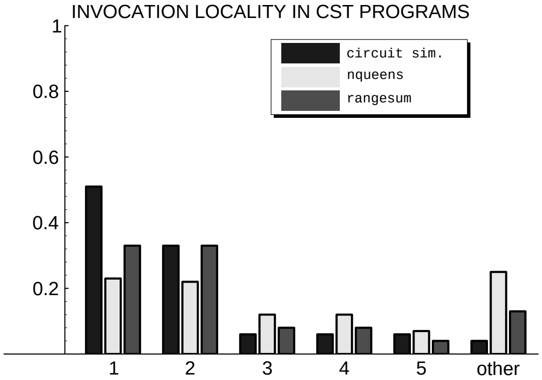
Invocation Locality in ABCL:

Application	*# msgs*	*% of msgs to i^{th} neighbor*			
		1	2	3	rest
nbody_I	273,157	0.29	0.20	0.15	0.35
nbody_B	463,722	0.30	0.21	0.15	0.33
parser	1,610	0.70	0.06	0.05	0.20
event sim.	94,965	0.54	0.20	0.17	0.09



Invocation Locality in CST:

Application	*# msgs*	*% of msgs to i^{th} neighbor*			
		1	2	3	rest
circuit sim.	14,909	0.51	0.33	0.06	0.16
nqueens	68,887	0.23	0.22	0.12	0.43
rangesum	6,003	0.33	0.33	0.08	0.15



Invocation Locality in CA:

Application	*# msgs*	*% of msgs to i^{th} neighbor*			
		1	2	3	rest
logic sim.	219,788	0.42	0.26	0.13	0.19
multigrid	582,913	0.33	0.19	0.10	0.38
pcbrouter	116,883	0.26	0.10	0.06	0.59

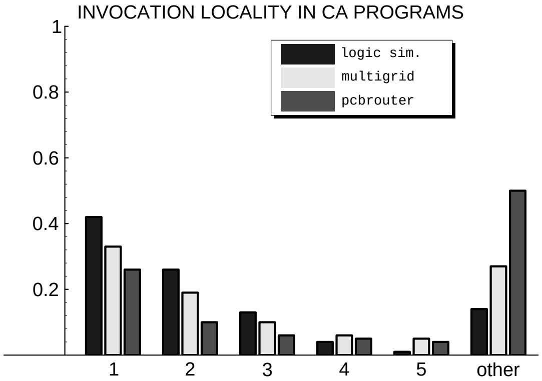


Figure 4: Invocation Locality in Object-Oriented Programs.

described here exploit the structured access and control-flow information available in concurrent object-oriented languages. Our approach focuses on identifying several common compositional structures for objects. If these structures can be identified, execution relating to each part can often be co-located or merged together, increasing the effective grain size. Some of the most promising opportunities for optimization are present with static object relations and recursive data structures. We discuss specialized analysis and transformation techniques for these cases which exploit the object-oriented expression of the program.

3.1 Static Object Relations

When static relationships are present between objects in some part of the computation, transformations can be applied at compile-time to increase the execution grain-size. Our analysis focuses on finding static relationships between an object and one of its state variables which are initiated at object-creation time. We limit our analysis to situations where the state variable is assigned an object in the initializer of the parent class. Two cases exist: (i) the child (state variable) object is *explicitly created in the initializer* of the parent class, and (ii) it is *created outside the initializer* of the parent class and a reference is passed into the initializer of the parent class.

To simplify the discussion, we define the following terminology:

write-once variable: An object state variable is write-once if there exists only one assignment to that state variable over the entire execution of a program.

internal variable: An object state variable is internal to a class if no statement in any of the methods exports any *references to the state of the variable* outside the class methods. Thus, given the statement,

$$\texttt{result = var} \rightarrow \texttt{meth1}(\ldots)$$

all references to the state variable `var` must satisfy one of the following constraints:

- `meth1` does not return any reference to the state of `var` whenever `meth1` returns a result.
- `meth1` returns a reference to the state of `var`, and `result` is a local state variable of the object which is *internal* to `var`.

external variable: An object state variable that is not internal.

3.1.1 Created in the Initializer

The code fragment shown below illustrates subcases where static relations can be identified and optimized. We adopt a C++-like syntax.

```
class A {
  ...; field b; ...

  initMeth(...) {
    :
    b = new B(...);
    :
  }
  Meth1(...); Meth2(...); ...; MethN(...);
}
```

Write-Once and Internal: If `b` is a *write-once and internal* variable, there exists a static object relation between the parent object of class `A` and the child object `b`. Note that neither global control-flow nor data-flow information is required since local control-flow information in the initialization code allows us to determine that the field `b` is assigned with a reference to a newly created object, and a class-wide analysis of all the methods for `A` allows us to determine that the relation is a static one. Co-locating `b` with the parent object of class `A` enables replacing all remote invocations from class `A` methods to class `B` methods with local invocations (i.e., ordinary procedure calls).

Write-Once and External: If `b` is a *write-once and external* variable, there exists a static object relation between the parent object of class `A` and the child object `b`. Co-locating `b` with the parent object of class `A` reduces invocation overhead between the pair; however, this may not translate to an overall improvement in the run-time overhead since references to `b` are being exported.

Write-Many and Internal: If `b` is a *write-many and internal* variable, the field `b` may be written to many times, and only a *temporary* static relationship exists between the parent and child objects. Although the child object is not present for the entire lifetime of the parent object, co-locating the parent and child objects for the duration of the child object's lifetime results in a reduction of remote invocations.

3.1.2 Created Outside the Initializer

In the code fragment shown below, an object of class `B` is created externally and then passed in as a parameter to the initializer for class `A`. This type of programming idiom is common in the use of objects as polymorphic containers such as lists and sets and creates static relationships as shown in Figure 5.

```
outer_method(...) {                class A {
    :                                  ...; field b ; ...
    c = new B;
    f = new A(..., c, ...);            initMeth(..., d, ...) {
    :                                      :
}                                          b = d;
                                           :
                                       }
                                       Meth1(...); Meth2(...); ..., methN(...);
                                   }
```

Two subcases exist:

Private: If `outer_method()` does not use `c` except as shown, then the child object of class `B` is effectively *private* to class `A`. Thus, the creation of the child object of class `B` can be pushed inside the initializer for class `A`. Because the container object of class `A` may be used for several different types, it must be type-split[7], and new specialized versions of the container (parent) class code must be created. As a result, each type-split version of the container (parent) class becomes a possible site for the optimizations discussed in Section 3.1.1.

Shared: If (*i*) `outer_method()` has other references to `c`, or (*ii*) `c` is a state variable for the class containing `outer_method()`, and `c` is referenced in the other methods, then its creation cannot be pushed into the initializer code for class `A`. However, the parent class `A` can still be type-split, and all remote invocations to the child object can be replaced with local invocations.

Figure 5: A polymorphic container, used at two points in a program. For any particular object, it contains a reference to only one object type.

```
t = type of current class                 Traversal Directions
F = {fields for the class}                R = { }
C = {code bodies of the class}            for each f in F do
R = {recursive fields}                        for each c in C do
E = {code bodies with                             if c contains a recursive
     extension calls}                                call on f
                                                     then insert(f,R)
Type Inference                                end
for each class do                         end
    based on the initialization
    code and other assignments,           Extension Calls
    compute the approximate               for each c in C do
    types of all fields.                      if c contains an allocation of t
                                                 then insert(c,E)
                                          end
```

Figure 6: Finding Recursive Data Structures

3.2 Recursive Data Structures

Exploiting class-level structure in object-oriented programs allows us to identify recursive data structures such as lists and trees[10] and transform their grain size. Typical implementations of recursive data structures in Actor languages localize the interesting control-flow information in the methods for one class. Consequently, we can analyze and optimize recursive data structures by the following steps: (*i*) examine the object code to identify recursive data structures, (*ii*) identify recursive axes which are traversed, (*iii*) identify allocation and extension points in the code, (*iv*) use the information to choose a representation which merges (one or more) objects along a traversal direction, yielding a larger grain size and finally, (*v*) transform the code to reflect the new representation. An outline of this analysis is sketched in Figure 6.

For example, in a set implementation based on a list of pairs, the code will contain the traversal direction, extension, and truncation operations on the list which can be identified by a class-wide analysis. Repeated transformation by grouping objects along a traversal direction gives rapid increase in grain size. The program may be transformed along several recursive axes simultaneously when there are multiple recursive fields, as in a binary tree. In Figure 7, the effects of transforming a list and a binary tree are illustrated. The binary tree has been transformed twice along the left-child axis. Subsequent transforming along the right child axis would merge each macro-object with two others.

Program Transformation: After identifying the recursive axes and the degrees of trans-

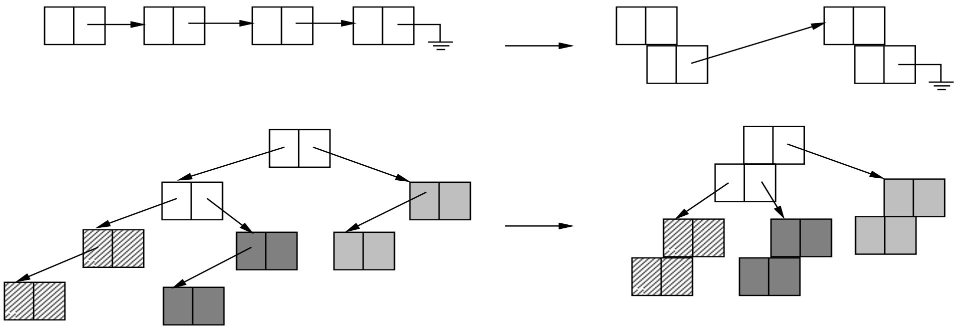


Figure 7: The basic and transformed representations of a recursive list and a binary tree. Modifications to the structures can be handled by preserved pointers.

formation, we must transform the code to reflect these decisions. The transformation steps are (i) color the object instances co-located in the new representation, and (ii) specialize method code for each color by type-splitting.

Object grouping can be achieved by merging the allocation points and producing place-dependent code for each member. The increase in code size is, in general, moderated by the fact that all internal group members reuse the same code. Thus, a transformed list would require two code versions: one for the internal objects in a group, and one for the last one.

4 Prototype Implementation and Measurements

This section describes the implementation of a prototype of the grain-size tuning system, and the results of some initial experiments with it. The grain-size tuning system is based on a generic intermediate format which can be used with a variety of concurrent object-oriented languages.

4.1 Expressing Locality

The relative location of objects must be determined to deal with locality issues at compile time. In addition, to optimize locality, it must be possible to change the relative location of objects. Traditional approaches [11] define regular mappings from the index space of arrays to the set of memories. These *direct mappings* fully define the data placement in the machine and can be exploited to specialize code to that particular mapping. Direct mappings are inappropriate for fine-grained object systems because they they provide no control over the relative location of objects and leave the run-time system little flexibility. So, instead of using a system which specifies object placement precisely, we use a system which constrains the placement of objects to be co-located. A co-location constraint specifies that both the objects will be on the *same* computing node (i.e. local to each other). Co-location constraints produce a two-level locality model: objects that are guaranteed to be local and others that are not; the distinction about communication being required for remote objects is captured in the model. In addition, using a system of relative placement constraints also leaves the run-time system free to place and move objects.

Compile-time determination of co-location constraints (as happens with the transformations in Section 3) allows the compiler to optimize interactions between the objects, using cheaper access mechanisms.

An Example: The pseudo-code fragments below show the optimization of a static object relation. In this case, an object of class **A** is determined to have a static reference to an object of class **B**. In the code for the initial method in class **A**, the original and transformed code are as shown:

```
BEFORE    (SEND-MSG ''new'' <dest> <size> ''B'' ''random'' () <args*>)
          (MOVE (IVAR 0) <dest>)

AFTER     (SEND-MSG ''new'' <dest> <size> ''B'' ''co-locate'' ''self'' <args*>)
          (MOVE (IVAR 0) <dest>)
```

The first code segment allocates memory for an object of class **B** in a random location, initializes it, and then assigns its reference to instance variable 0, `(IVAR 0)`. The `''random''` annotation causes arbitrary placement of the object. The transformed code-fragment constrains the placement to be the same as that of the invoking object by using a `''co-locate'' ''self''` annotation. To exploit this co-location, the compiler specializes all invocations on `(IVAR 0)` to be of an inexpensive, local variety. Local invocations can then be compiled as ordinary procedure calls; no message passing is required.

4.2 Implementation and Performance Results

Our prototype compiler implements the analysis and transformation of static object relations described in Section 3.1. We use two simple programs and two application benchmarks written in Concurrent Aggregates to examine the performance of our grain-size tuning system.

- **Slope Finder** Program to determine slopes of lines constructed from pairs of points.

- **Tree Sum** Program to sum leaf values in a tree of objects.
- **Logic Simulator** An event-driven logic simulation of a static object network managed through a concurrent priority queue.
- **Printed Circuit Board Router** Concurrent A$*$ search to route nets around rectangular obstacles on a printed circuit board.

The improvement resulting from transforming a remote invocation to a local one depends on a wide variety of machine and system specific parameters.[5] Therefore, we characterize the grain-size benefits by showing how often we can apply our transformations and what fraction of the full-cost, remote invocations can be removed from a program. Table 1 shows the number of static optimization points found, the number of messages sent in the unoptimized and optimized cases, the fraction of communication traffic eliminated, and the estimated increase in grain-size for each program. The increase in grain-size is estimated based on the reduction in communication traffic.[6]

Program Name	*Optimized Points*	*Msgs Sent in Unopt.*	*Msgs Sent in Opt.*	*Reduction in Comm.*	*Grain-Size Increase*
Slope Finder	2	73	41	43.84%	1.78
Tree Sum	5	1164	916	21.31%	1.27
Logic Simulator	28	521,093	475,405	8.77%	1.10
PC Board Router	1	89,554	80,495	10.12%	1.11

Table 1: Performance of the Grain-Size Tuning System.

The grain-size tuning system finds static object relationships in all the programs. In the slope finder program, each line object uses two point objects both of which have a static relationship with the line. The tree sum program has static links for communication between a node and its parent and child nodes. Co-locating the objects along the vertical dimension of the tree reduces the cost of this communication. In the logic simulator program, remote invocations between the gates and the circuit nodes can be replaced with a local version. While the communication reduction in this case is modest, the large number of optimization points found in the logic simulator program encourages us that static optimizations may turn out to be a significant contributor to our grain-size tuning system. In the printed circuit board router program, each node on the board grid has a fixed relationship to its corresponding (x,y) point. Thus, remote invocations from a node to a point can be replaced with local invocations.

[5]The cost of a remote invocation can vary from in several milliseconds in systems running OSF/1 to a few microseconds in the highly tuned J-machine.

[6]If the amount of work is conserved, the grain size can be found by dividing the work by the number of messages.

5 Related Work

Unlike the approaches for specializing invocations in sequential object-oriented languages which reduce the cost of type-dependent polymorphic dispatches[9, 7], our approach specializes invocations so as to reduce communication and overhead due to message passing.

Two bodies of work, both of which simultaneously optimize data placement and execution grain size, are similar to ours. First, the compiler for the MasPar MP-1[6] lumps together operations on a number of array elements and allocates these chunks of work to individual processors. Second, efficient execution of concurrent logic languages has been obtained by grouping successive elements of a stream[14]. The first approach is typical to data-parallel programming languages, while our approach and the stream approach can work with more general heterogenous data structures. Grain-size tuning is an issue even in shared memory machines[1] since large grains are required to achieve reasonable execution efficiency.

6 Summary

Our work focuses on making the execution of fine-grained concurrent object-oriented programs efficient. The key to our approach is to transform the *execution grain size* of programs to match the underlying hardware.

Invocation traces of concurrent object-oriented programs show the existence of significant invocation locality. Large improvements in invocation and message-passing overhead are possible by proper exploitation of the observed locality.

Global control flow and data flow analysis is quite difficult in concurrent object-oriented languages because of the pervasive use of type-dependent dispatch and dynamic storage allocation. Without this information, we are forced to rely on program structure. We describe several analyses that identify static object relations and candidates for optimization by exploiting initialization information and the class structure of object-oriented programs. Additional transformations exploit the nature of recursive data structures.

The work described in this paper is part of the Concert project whose goal is to achieve efficient, portable, and scalable execution of concurrent object-oriented languages via grain-size tuning techniques. Our current system supports execution of Concurrent Aggregates programs on both a uniprocessor simulation environment and a parallel implementation on the CM-5. Current efforts focus on developing better techniques for type-inference, aliasing analysis and the dynamic detection of invocation locality.

References

[1] A. Agarwal, B. Lim, D. Kranz, and J. Kubiatowicz. April: A processor architecture for multiprocessing. In *International Symposium on Computer Architecture*, 1990.

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

[3] P. America. Pool-T: A parallel object-oriented language. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199–220. MIT Press, 1987.

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

[5] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, SE-13(1):65–76, January 1987.

[6] T. Blank. The Maspar MP-1 architecture. In *Proceedings of COMPCON*, pages 20–4. IEEE, 1990.

[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] A. A. Chien and W. J. Dally. Concurrent Aggregates (CA). In *Proceedings of Second Symposium on Principles and Practice of Parallel Programming.* ACM, March 1990.

[9] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Eleventh Symposium on Principles of Programming Languages*, pages 297–302. ACM, 1984.

[10] L. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Computing*, 1(1):35–47, 1990.

[11] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines. In *Supercomputing '91*, pages 86–100, Nov. 1991.

[12] 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.

[13] E. Myers. A precise interprocedural data flow algorithm. In *Seventh Symposium on Principles of Programming Languages*, pages 219–30, 1980.

[14] V. Saraswat, K. Kahn, and J. Levy. Janus: A step towards distributed constraint programming. In *Proceedings of the North American Conference on Logic Programming*, Austin, Texas, October 1990.

[15] K. Smith and R. Smith II. The Experimental Systems Project at the Microelectronics and Computer Technology Corporation. In *Proceedings of the Fourth Conference on Hypercube Computers*, 1989.

[16] C. Tomlinson, M. Scheevel, and V. Singh. Report on Rosette 1.0. MCC Internal Report, Object-Based Concurrent Systems Project, December 1989.

[17] W. E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Seventh Symposium on Principles of Programming Languages*, pages 83–94, 1980.

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