# The Concert System – Compiler and Runtime Support for Efficient, Fine-Grained Concurrent Object-Oriented Programs

Andrew A. Chien Vijay Karamcheti John Plevyak
Department of Computer Science
1304 W. Springfield Avenue
Urbana, IL 61801
{achien,vijayk,jplevyak}@cs.uiuc.edu

June 11, 1993

**Abstract**

The introduction of concurrency complicates the already difficult task of large-scale programming. Concurrent object-oriented languages provide a mechanism, encapsulation, for managing the increased complexity of large-scale concurrent programs, thereby reducing the difficulty of large scale concurrent programming. In particular, fine-grained object-oriented approaches provide modularity through encapsulation while exposing large degrees of concurrency. Though fine-grained concurrent object-oriented languages are attractive from a programming perspective, they have historically suffered from poor efficiency.

The goal of the Concert project is to develop portable, efficient implementations of fine-grained concurrent object-oriented languages. Our approach incorporates careful program analysis and information management at every stage from the compiler to the runtime system. In this document, we outline the basic elements of the Concert approach. In particular, we discuss program analyses, program transformations, their potential payoff, and how they will be embodied in the Concert system. Initial performance results and specific plans for demonstrations and system development are also detailed.

**Indexing Keywords:** Concurrent object-oriented languages, Multicomputers, Parallel Computing, Compilation, Runtime Systems.

## 1 Introduction

In the sequential computing world, the increasing complexity of software applications, systems, and requirements places growing burdens on software developers, taxing their ability to manage complexity. To a great extent, this problem is responsible for the rapidly increasing popularity of object-oriented programming techniques. The encapsulation provided by such techniques supports code reuse and separate design (modularity), enabling the construction of larger, more complex systems more rapidly than previously possible.

In the parallel computing world, traditional software development challenges are complicated by concurrency and distribution. These additional concerns exacerbate the need for program modularity to manage complexity. Concurrent object-oriented languages support program modularity through object encapsulation, allowing the hiding of concurrency, distribution, or other irrelevant detail. Further, *fine-grained* concurrent object-oriented languages both provide encapsulation and expose large scale concurrency in application programs, and therefore are a promising approach to parallel programming.

The primary advantage of concurrent object-oriented languages is encapsulation, a critical tool for managing the complexity of large programs. Modularity tools allows the exploitation of concurrency whether it be homogeneous and heterogeneous, or expressed in as data parallelism or task parallelism. Exploitation of diverse concurrency structures is possible because encapsulation can confine of each subcomputation, allowing the programmer to reason about them separately. For example, in a particle-in-cell code, a partial differential equation solver and a graph algorithm which computes the mapping of particles for the next time step can easily be written to run concurrently. Expressing equivalent concurrency in a data parallel model is quite difficult.

While attractive for programmability reasons, the primary drawback of fine-grained, concurrent object-oriented languages to date has been their inefficiency (compared to their competitors such as parallel FORTRAN dialects). In addition, the most efficient implementations of such languages have relied on specialized hardware to achieve high performance [15, 36, 42]. The primary goal of the Concert project is to develop compiler and runtime techniques to make fine-grained concurrent object-oriented languages portable and efficient. By portable and efficient, we mean that the programs should run efficiently both on uniprocessors and on parallel computers built from stock microprocessors.

While ideally, our implementation techniques should apply to arbitrary multiple-instruction multiple-data (MIMD) parallel machines, a reasonable standard of portability to demonstrate the generality of the techniques is based on the following assumptions about the basic structure of future parallel machines. We assume processing nodes based on workstation or PC-like architectures, fast local processing with a deep memory hierarchy (two or more levels), inexpensive access to a communication network (no operating system calls required), and a high bandwidth communication network. In particular, we make no assumptions about special mechanisms for local synchronization, global synchronization, or a global address space. This model is compatible with a broad range of existing and announced multicomputers [39, 13, 12]. In such machines, the fundamental performance issues are balancing the level of concurrency exploited against the cost of scheduling and context switching, exploiting data locality within and between nodes and achieving a reasonably balanced work distribution throughout the machine.

In the Concert system, our approach focuses on execution grain size tuning. We define a program's *execution granularity* as the size of fragments of computation that are scheduled independently at runtime. In the execution of a parallel program, grain size determines the processor efficiency of the application[1] as shown in Figure 1. For a particular machine target and runtime system implementation, the cost of communication, concurrency control, and scheduling can be viewed as fixed, and processor efficiency is determined by the ratio of the amount of work in each computational grain to the overhead. Thus, executing a program with appropriate grain size is essential for high performance; too large a grain size produces concurrency-limited execution, too small a grain size produces an overhead-dominated computation.

In concurrent object-oriented programs (COOP), each object invocation can be viewed as an execution grain. Tuning the execution grain size therefore implies partitioning or merging object invocations together into a single execution grain. Since in most cases, the goal is to increase the grain size, we focus on merging the object invocations. Consider the elements of an object invocation: name translation, message formatting, transmission, reception, buffering, scheduling, and dispatch. A remote procedure call (RPC) style call-return is shown in Figure 1. For the return, many of the steps are repeated. Since a typical object invocation may be only 10-20

[1]This ignores idle time.

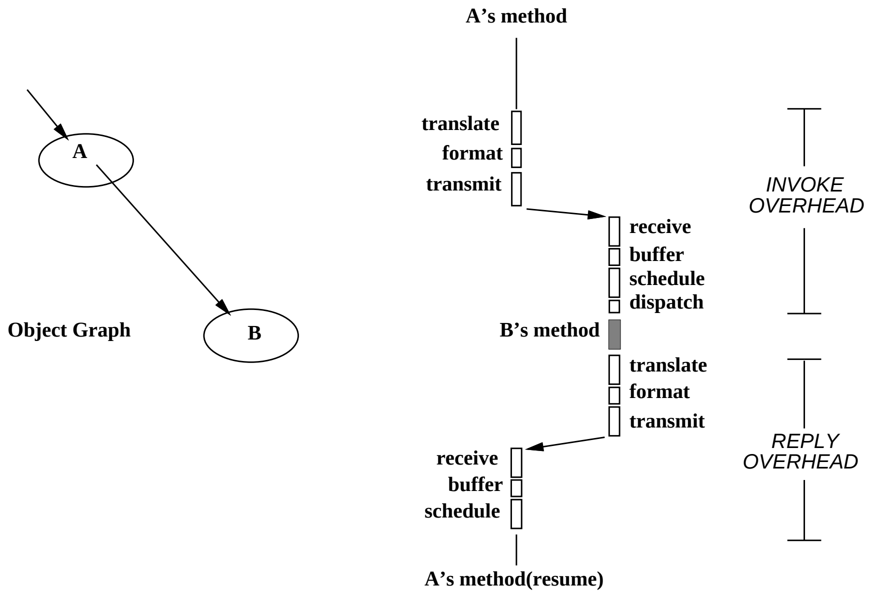


Figure 1: An invocation sequence between objects **A** and **B**.

instructions, the overhead for the object invocation steps can dominate the actual work (shown in gray). In comparison, an analogous sequential program might require a dozen or so instructions as overhead for a procedure call and return, with arguments and return values passed on the stack. If optimized, procedure call overhead can be reduced further, with parameters passed in registers or the with the procedure being inlined. To achieve competitive efficiency, an implementation of a concurrent object-oriented language must use the cheapest mechanisms possible for an invocation, and maximize the opportunities for using them. To reduce the cost of invocation, a variety of type, location, and concurrency information is helpful, enabling the elimination of message formatting, scheduling, and dispatch overhead. This information may be available at compile time, or only at run time. Precisely when it becomes available has direct impact on how it can be exploited and the impact it has on program efficiency. These issues are discussed in greater detail in Section 3.

The Concert system provides a framework for systematically extracting and exploiting the necessary information for grain size tuning in fine-grained, concurrent object-oriented programs. Such a framework requires an integrated approach involving collaboration between the compiler and runtime system.[2] The Concert system involves four basic types of techniques for increasing execution grain size: compile-time optimization, compile-time speculative transformation and optimization, compiler-guided runtime optimization, and high performance runtime systems. The critical issues and challenges for each of these types of optimization are discussed in Section 3.

As a general rule, the Concert system exploits information as soon as it is available, enabling a broader scope of optimization and reducing the cost of program transformation than would be possible at a later stage. In concurrent object-oriented languages, the requisite information may only become available at any phase from compile to run time, so optimization at all phases is essential to high performance. For example, it would be preferable to do all optimization at compile-time, but due to limitations of static analysis in programs with implicit typing, dynamic

[2]Hence the name *Concert*, indicating a concerted effort of both the compiler and runtime.

allocation, and pervasive use of references, the requisite information may not be available. If partial information is available, the Concert compiler might choose to compile specialized versions of the program and select amongst them at run time. And, if the compiler cannot statically narrow the possibilities, it will resort to instrumenting the program for dynamic compilation, specializing the program code with respect to the structure and distribution of program data structures as information becomes available. If dynamic compilation is not feasible, the Concert system will rely on clever scheduling to increase execution grain size and efficient runtime operations to execute grains as efficiently as possible.

We are building a high performance concurrent object-oriented system, called the Concert system, based on these ideas both as a demonstration and an experimental vehicle for exploring these techniques. The Concert system embodies the compiler and runtime support required to optimize programs across the full range of stages.

**Overview** The remaining sections of this document are organized as follows. Section 2 discusses the basic language model being used for programming, analysis, and optimization. The model is a simple concurrent object-oriented language with single inheritance. In Section 3, we detail the four basic means of exploiting program information to tune execution grain size. In particular, we discuss the potential benefits of several types of optimizations with respect to an actual runtime implementation. Sections 4 discusses how the ideas behind the Concert system will be evaluated and some of the questions we are studying. Section 5 discusses the related work, particularly data parallel approaches, contrasting them to the approach outlined here. Finally, Section 6 summarizes the paper and the current status of the project.

## 2 Basic Language Model

Compiler analysis and optimization must take place in the context of a programming language semantics and execution model. Because our goal is to develop generic analysis and optimization techniques, we have chosen to study a simple concurrent object model, incorporating the essential features of concurrency and object-orientation, and optimize it for a generic model of parallel machines, based on stock hardware (see Section 1). Using generic models will allow the optimization techniques to be easily transported to other concurrent object-oriented systems, particularly those based on broadly accepted standards.[3]

Concurrent object-oriented programs (COOP) express computation as a set of autonomous communicating entities (objects). Generally, these objects reside in a globally shared object namespace. Each object is single threaded and communicates by asynchronous message passing (invocations). See Figure 2. Data is encapsulated within objects and can only be accessed through messages. Synchronization constraints are enforced by selective processing of messages. Objects may send messages, create additional objects, and modify their local state in response to a message. A theoretical basis for reasoning about concurrent object oriented programs can be found in [1]. In summary, the programmer specifies only the essential aspects of the computation: concurrency control and enabling of computations, but does not manage location, storage, or detailed scheduling explicitly. These tasks are left to the compiler and runtime system.[4]

[3]For example, our model is general enough that the optimization techniques will apply to concurrent languages based on C++, including languages with explicit thread creation.

[4]For regular, numeric computations, we may consider the addition of some simple data decomposition operations. However, for other types of applications, explicit locality control is difficult to use effectively.

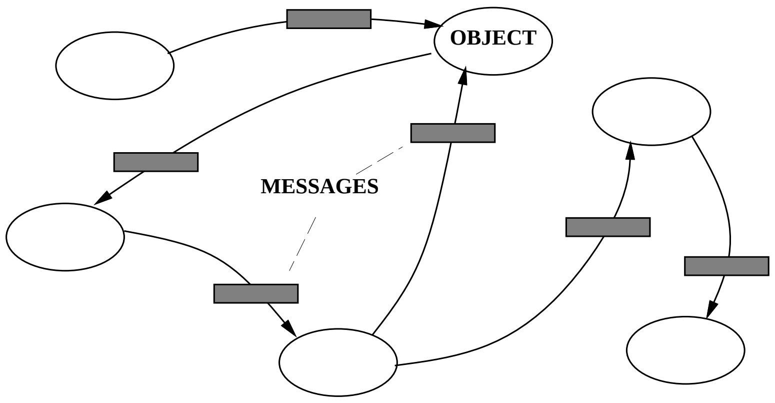


Figure 2: Autonomous Communicating Objects

The Concert system analyses and optimizes programs which conform to this basic concurrent object-oriented model, so the techniques developed should be applicable to a wide range of concurrent object oriented languages. In practice, this is ensured by defining all optimizations and transformations with respect to a generic compiler intermediate form for concurrent object-oriented languages. Any language that can be translated to that intermediate form can be optimized by the compiler. In fact, we are considering building compiler front-ends for several different source languages. This would demonstrate the genericity of the optimizations, as well as allow direct comparisons isolating the cost of programming language features.

The specific language we are supporting initially is an extended version of Concurrent Aggregates (CA) [11]. As with most concurrent object-oriented languages, CA augments the basic asynchronous message passing model with common idioms (such as RPC and tail forwarding [21]), inheritance (a mechanism for code reuse), and some concurrency control constructs. Other novel aspects of Concurrent Aggregates include aggregates (parallel collections), and meta-level structures for parallel composition, first class messages and continuations. This range of features in CA ensures that our intermediate form can support a broad range of concurrent object-oriented languages with a variety of language features.

Because each object invocation is a potential source of concurrency, a concurrent object oriented program can be executed in parallel by distributing the objects over the processors in a distributed memory machine and using message passing for inter-object communication. The basic approach gives rise to high levels of fine-grained concurrency. Because this fine-grained concurrency typically cannot be exploited efficiently on stock hardware, the system must collect invocations into groups which can be efficiently scheduled and executed on the target platform. Automatically achieving this grouping and thereby achieving efficient execution is the goal of the Concert project.

## 3 Concert Approach

The Concert system embodies an integrated approach to achieving efficient execution of fine-grained, concurrent object-oriented programs. Because most parallel machines cannot support fine-grained concurrency efficiently, execution grain-size must be increased to acceptable levels. Though encapsulation, polymorphism, and dynamic allocation are desirable from a programming perspective, they make static analysis and transformation of programs extremely challenging. And because object-oriented programs encourage many small procedures, the resulting high frequency of type-dependent procedure call implies that many call sites must be optimized to obtain sizeable increases in grain size. The most important information for optimization is type and locality information. To maximize the exploitation of such information whenever it becomes available, the Concert system employs a four tiered approach to obtaining and exploiting program information; each tier supports the exploitation of different levels of information that become available at a different stage of program compilation or execution. Such an approach inherently requires close cooperation between the compiler and runtime system. This structure allows program information to be exploited at an appropriate stage (typically as soon as possible) and conserves information, supporting program transformation both at compile and run time (as shown in Figure 3).

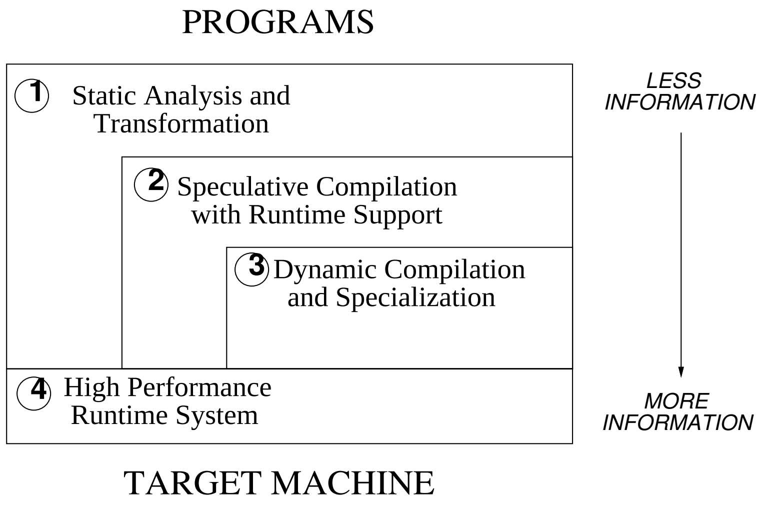

PROGRAMS
1 Static Analysis and Transformation
2 Speculative Compilation with Runtime Support
3 Dynamic Compilation and Specialization
4 High Performance Runtime System
LESS INFORMATION
MORE INFORMATION
TARGET MACHINE

Figure 3: Four-tiered Structure of the Concert System

The four tiers of the Concert system span the range from compile to run time. The first tier is static program analysis and optimization which happens at compile time. Analysis of object-oriented programs is particularly challenging because the control flow depends on type information which in turn depends on data flow. The introduction of concurrency and distribution further complicates the situation as ideally the compiler would manage both concurrency control and locality in the machine to achieve high performance. In general, static optimizations require precise information to assure their safety and thereby correct program implementation. The second tier is speculative optimization based on imprecise information about program structure. This level of optimization is useful when analyses indicate a particular structure is likely, but cannot prove it for all possible program executions. Because a static transformation based on the particular structure would be unsafe, the improved code must explicitly test for the structure. The resulting transformation is speculative because it does not always result in improved execution speed. The third tier is dynamic compilation based on local runtime information about program structure. Dynamic compilation can be based on precise information unavailable at compile time, thereby producing much more efficient code. However, this increased efficiency must be weighed against the high cost of runtime compilation. The fourth tier, which supports the other three tiers, is a high performance runtime. But beyond providing fast basic primitives, the runtime interface must allow optimizations to be expressed, support speculative optimization, support dynamic compilation, and provide a variety of more traditional services such as efficient scheduling and resource management.

## 3.1 Static Analysis and Optimization

The first tier in the Concert system is static analysis and optimization. Static optimization is attractive because it has no runtime cost and can be based on global program information. If precise static analysis is available, then the compiler can use the cheapest mechanisms available for a computation. For example, if the object type and the assurance of locality can be inferred from analysis, then method invocations on that object can be inlined, dramatically improving execution efficiency.

In addition to classical analyses used in languages such as FORTRAN, static analysis of concurrent object-oriented programs has two distinctive steps, type inference and structure analysis. Type inference is necessary to determine the control flow structure of a program because object-oriented programs use type dependent dispatch for modularity and polymorphism. Structure analysis is necessary to determine data flow structure because most object-oriented languages make extensive use of dynamic storage allocation and references. We discuss the two distinctive analyses, type inference and structure analysis, in the following sections. Subsequently, we discuss how the resulting program information could be used to optimize programs.

### 3.1.1 Type Inference

Type inference uses the structure of a program to deduce the type (or range of types) that program variables can take on. Type information can be used both directly in the generation of efficient code and in determining control flow of an object-oriented program. For example, in Figure 4 the type of $n$ is known to be be $ClassA$ and the type of $self$ can be resolved to $Main$. Consequently, invocations $foo$ and $bar$ can be optimized to procedure calls, eliminating the overhead of a dynamic dispatch. The Concert compiler is capable of this sort of automatic analysis and optimization.

```
class ClassA
  method bar
    1
  end method
end ClassA

class Main
  var n
  method run
    n := self foo: (ClassA new);
    print (n bar) + 1
  end method
  method foo: a
    a
  end method
end Main

(Main new) run
```

```
class Main
  var n
  method run
    n := (ClassA new);
    print 2
  end method
end Main
```

Figure 4: Type Inference Example

The Concert compiler uses a constraint based approach to type inference which determines not only whether a program is type safe but also a safe approximation of the type of each program variable. The type inference system is an extension of that described in [29]. Local constraints are established and propagated according to a set of rules for primitive operations such as object creation, assignment, and usage. A continuous approximation of the control flow is maintained, and constraints are built for the paths along this flow. When all constraints induced by the program have been added, their solution is a safe approximation of both program variable types and control flow. Because the solution is only subject to program structure, for some program variables precise type information may not be available. The available type information is used to determine the interprocedural control flow graph required for a variety of traditional program analyses.

The Concert type inference system extends the capability of of traditional constraint based type systems for object-oriented languages [29] in four ways. First, the extended inference system allows variable precision based on control flow and object creation points. This allows compiler effort to be focused where it will be most productive. Second, in order to support a broad variety of concurrency constructs, the Concert type system supports the analysis of programs with first class selectors, continuations, and messages. Third, to ensure compilation of all working programs, our type system handles typing failures gracefully, reporting them to the programmer and generating sufficient runtime checks to ensure correct program execution. Finally, the type information produced by inference is stored in a flow-sensitive database. This organization is particularly helpful for dependent program transformations.

### 3.1.2 Structure Analysis

Structure analysis is used to approximate runtime data structures in programs with dynamic allocation. Structure analysis is important in concurrent object-oriented programs, as such programs tend to make extensive use of dynamic allocation and sophisticated data structures. The information derived from structure analysis is essential to high quality optimization and is typically used to compute aliasing relationships, data flow, and storage structure. This information can be used to reorder computations and to improve program locality. For example, information that a runtime data structure represents a singly-linked list can be used to group objects ensuring that references between them are local.

Structure analysis interprets the program against a model of the program store, producing a conservative approximation of the effect of the program. The resulting store approximation captures the structure of the heap. However, because the representation of the heap must be finite, and the execution of the program is potentially infinite, the heap structure is summarized to capture structures immediately reachable from the program values at each program point. Such local information is sufficient for a variety of grain size optimizations.

The Concert system uses a structure analysis based on Abstract Storage Graphs (ASG's) which is capable of precise analysis of complex dynamic structures such as singly and doubly linked lists, object hierarchies, and octrees. The Abstract Storage Graph [30] is an enhancement of Chase's Storage Shape Graph (SSG) [8], which adds distinct node and reference types as well as identity paths to achieve more precise analysis. As with SSG's, the analysis is based on a data flow framework. Within the framework, we define the lattice of solutions to be an abstraction of the program store. The transfer functions are then the abstract interpretation of the program. The meet operator is a safe merge of these storage approximations.

An example of the information that Concert's structure analysis is able to derive is shown in Figure 5. Analysis of the program on the left produces the abstract storage graph on the right. The graph compactly represents a singly linked list of indefinite length. The node in the center is

```
x = nil;
loop
  y = Node new;
  y set_next: x;
  x = y
end loop
```

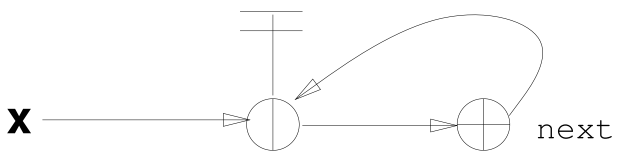


Figure 5: Structure Analysis Example

a distinct choice node and the node at the right summarizes all the pair elements in the list. Each node in the list either points to NIL or another list node. An example of optimization based on structure information is given in Section 3.1.3.

### 3.1.3 Exploiting Static Information

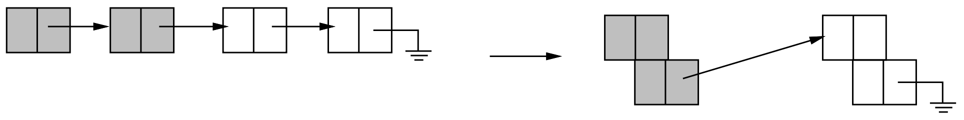
Type inference and structure analysis coupled with traditional program analyses produce a variety of information useful for optimizing concurrent object-oriented programs. In particular, we focus on optimizing method invocations, as these enable a wealth of more traditional optimizations. However, because the Concert system is intended for parallel machines with distributed memories, locality optimization is a prerequisite to more traditional optimizations on method invocations which presume locality.[5] Once locality is ensured, type information can be used to resolve type-dependent dispatch statically, optimize procedure linkage, or inline procedure calls.

Storage structure information can be used to compel locality, enabling a wealth of program optimizations. For example, consider the structure analysis shown in Figure 5 that identifies a singly-linked list. To enhance locality, lists can be *tiled* (see Figure 6) to increase the number of local invocations along its spine. Tiling two pairs together increases execution grain size by allowing a pair of operations on adjacent white and adjacent gray list objects to be scheduled as a single computation grain. The code can be optimized as shown in Figure 6, converting the message send to a procedure invocation, then inlining the procedure invocation and accessing the state of both objects directly. Tiling is a special case of more general object fusion and clustering operations which force locality thereby enabling other program optimizations.

Object merging and grouping optimizations can reduce execution overhead significantly. Tiling pairs of elements in a list can produce a 50% reduction in overhead. Consider that in the Concert runtime [25], the most general message send operation requires $44.2\mu s$, and a simple procedure call requires only $0.15\mu s$. This means traversing each pair of list elements requires $88.4\mu s$ for the `List1` implementation, $44.35\mu s$ for the `List2` implementation, and $44.2\mu s$ for the `List3` implementation. Going from `List2` to `List3` requires type information and procedure inlining. Though the benefit of these optimizations are small in this case, it is greater in general if the method bodies are larger. Not only does inlining eliminate message send overhead, it increases the procedure size, increasing the effectiveness of traditional compiler optimizations such as common subexpression elimination and instruction scheduling. This is particularly important in object-oriented programs which tend to have high procedure-call frequencies.

In this section, we described optimizations which presumed precise information. However, in many cases, such precise information is not available. In subsequent sections, we examine how the Concert system exploits even imprecise information to improve program execution.

[5]Virtually all traditional program optimizations presume a shared address space and make no allowance for managing memory locality.



```
class List
  var v, next
  method eval: sum
    next eval:
      (v + sum)
  end method
end List
```

```
class List2
  var v1, v2, next
  method eval2: sum
    next eval:
      (v2 + sum)
  end method
  method eval: sum
    self eval2:
      (sum + v1)
  end method
end List2
```

```
class List3
  var v1, v2, next
  method eval: sum
    next eval:
      (v1 + v2 + sum)
  end method
end List3
```

Figure 6: Tiling pairs of elements in a List.

## 3.2 Speculative Transformation and Runtime Support

The second tier of the Concert system is speculative program transformation. When precise information is not available, the compiler cannot safely optimize the program. If imprecise information is available, it may be possible to optimize for the likely cases, selecting amongst several specialized versions at runtime. If the specialized versions capture actual program behavior, it is possible to obtain much of the benefit of static optimization. However, the disadvantages of this approach are the cost of runtime checks and significant increases in code size.

Speculative optimization is useful when static analyses produce imprecise information, narrowing the possibilities so that they can be quickly resolved by runtime checks. Specialized versions of code can be compiled for each of the alternatives, amortizing the cost of a runtime check over a large number of optimized operations. Because speculative transformation is done at compile time, the full range of optimizations can be applied to each specialized path, producing performance nearly as good as if precise information were available. Properties such as object type, location, and current status (active or dormant) can be used as a basis for speculative optimization[6]. For example, the type inference system may be able to narrow the type of a particular variable to a small set of types. Speculative compilation to insert runtime type checks and select amongst paths specialized for each type can enable a chain of optimizations along each path. Such optimizations can produce large savings in execution time.

Speculative transformations can be based on runtime checks, assertions, and hints, depending on the information available and the desired temporal and spatial scope of the optimization. All three types of speculative transformations require runtime support via special runtime operations to ensure their efficient implementation. We describe each type of speculative optimization and the required runtime primitives below.

[6]Location (local or remote) can also be considered part of an object's implementation type.

- **Runtime checks** are used to exploit fortuitous object properties to select runtime mechanisms at a particular program point. Requires runtime calls to check properties.
- **Assertions** extend the temporal and spatial extent of the object property, increasing the range of optimization. Requires runtime calls to make and revoke assertions.
- **Hints** attempt to influence the runtime system into improving malleable object properties such as location to improve performance. Requires runtime calls to suggest desirable object properties.

Runtime checks query the runtime system to determine actual object properties. On this basis, the cheapest possible runtime mechanisms can be selected at runtime. For example, choosing between stack-based or network-based method invocation sequences, depending on whether the object invocation target is local, could be optimized by runtime checks. While this approach can give significant benefits, it also incurs significant runtime overhead and cannot enable optimization across context switches. An example of runtime checks is shown in Figure 7 (b), where a series of invocations are made on an object, `B`, whose location is unavailable at compile time. Runtime checks allow procedure calls to be dynamically selected instead of general method invocations where appropriate. Runtime checks are required before each method invocation since the object might be migrated between invocations.

Assertions extend the temporal and spatial scope of information about an object property for the purposes of optimization. This allows the compiler to generate code which tests for an object property, asserts that property, then executes a long sequence of code optimized based on the assertion. Without assertions, to generate safe code, the compiler would have to assume that object properties could change at arbitrary times, limiting optimization. Instead, if the runtime system must invalidate an assertion, it invokes a callback function provided by the compiler, giving it the opportunity to revert to an unoptimized version. This is straightforward if the compiler simply uses compatible storage maps for all versions. An example of assertion based optimization is given in Figure 7 (c) where following the check for locality, `B` is asserted to be local. This allows all of the code in the true arm of the conditional to be optimized for local invocations – no additional checks are required. Conversion to procedure calls and even procedure inlining are both possible. If the specialized code section is large, eliminating checks can give significant benefits. The callback code is not shown.

Hints extend both runtime checks and assertions by attempting to influence malleable object properties such as locality. Hints tell the runtime that it would be beneficial if a particular property were true. This allows the compiler to express to the runtime system any assumptions it has used in performance optimizing the program. In general, hints can be used to guide policies of the runtime. Hints are not binding on the runtime, and the compiler must still assure correct program execution if hints are not respected by the runtime system. Figures 7 (d) and (e) show how the basic speculative transformations might be annotated with hints. The hints (`HINT(B,'local')` statements) suggest to the runtime that if `B` is not already local, it might be worthwhile to migrate it. Because hints are non-binding suggestions to the runtime, both example programs with hints will run correctly if all hints are removed.

Speculative transformations can yield significant benefits. Runtime checks as in Figure 7 (b) can reduce invocation overhead from $11.17\mu s$ for general local method invocation to $0.15\mu s$ for a local procedure call, a dramatic improvement which far outweighs the cost of the inline test. This means that runtime checks can benefit even single invocations. Assertions are particularly important since they enable interprocedural optimization and optimization across context switches.

```
...
INVOKE-METHOD(B, meth1)
... computation
INVOKE-METHOD(B, meth2)
... computation
INVOKE-METHOD(B, meth3)
...
```

**(a)**

```
...
if ( CHECK(B, ‘local’) )
  B→meth1;
else
  INVOKE-METHOD(B, meth1)
endif
... computation
if ( CHECK(B, ‘local’) )
  B→meth2;
else
  INVOKE-METHOD(B, meth2)
endif
... computation
if ( CHECK(B, ‘local’) )
  B→meth3;
else
  INVOKE-METHOD(B, meth3)
endif
...
```

**(b)**

```
...
if ( CHECK(B, ‘local’) )
  ASSERT(B, ‘local’)
  B→meth1;
  ... computation
  B→meth2;
  ... computation
  B→meth3;
  RETRACT(B, ‘local’)
else
  INVOKE-METHOD(B, meth1)
  ... computation
  INVOKE-METHOD(B, meth2)
  ... computation
  INVOKE-METHOD(B, meth3)
endif
...
```

**(c)**

```
...
HINT(B,‘local’)
if ( CHECK(B, ‘local’) )
  B→meth1;
else
  INVOKE-METHOD(B, meth1)
endif
... computation
if ( CHECK(B, ‘local’) )
  B→meth2;
else
  INVOKE-METHOD(B, meth2)
endif
... computation
if ( CHECK(B, ‘local’) )
  B→meth3;
else
  INVOKE-METHOD(B, meth3)
endif
```

**(d)**

```
...
HINT(B,‘local’)
if ( CHECK(B, ‘local’) )
  ASSERT(B, ‘local’)
  B→meth1;
  ... computation
  B→meth2;
  ... computation
  B→meth3;
  RETRACT(B, ‘local’)
else
  INVOKE-METHOD(B, meth1)
  ... computation
  INVOKE-METHOD(B, meth2)
  ... computation
  INVOKE-METHOD(B, meth3)
endif
...
```

**(e)**

Figure 7: Example code fragments showing different Speculative Transformations. INVOKE-METHOD() calls the network-based invocation routine while `obj→method` uses a local procedure call.

Since the cost of inserting and retracting assertions is comparable to a local invocation, assertion based transformations can begin to pay off even for a single method invocation. On the other hand, the cost of invalidating an assertion can be very high; thus, optimizations using assertions must be applied with discretion. Hint-based transformations such as affinity for a particular object can involve large costs to change object properties. Consequently, they appear to be worthwhile only when many objects on a particular node exhibit affinity for another object. The effectiveness of hints is currently unknown, but some related studies show that locality hints can improve performance [6].

While speculative optimizations can dramatically increase the opportunities for optimization, they are limited to cases where the possibilities can be narrowed and specialized code generated for each. When program structure depends strongly on input data or the evolution of computation, static analyses will be unable to infer even partial information. In other cases, the code size increase due to speculative techniques will limit their applicability. In the next section, we discuss techniques, dynamic compilation and specialization, which can deliver high performance in such cases.

## 3.3 Dynamic Compilation and Specialization

The third tier of the Concert system is dynamic compilation and specialization. When the static analysis system is unable to narrow the possibilities, the Concert system resorts to dynamic compilation, specializing on the basis of local information available at runtime. First, dynamic compilation is attractive when program behavior and data structures depend strongly on input data. Second, dynamic compilation is also attractive when a multiplication of properties on which to specialize causes a code size explosion in the speculative compilation approach. Finally, dynamic compilation is attractive when specialization is linked to properties that are varying periodically, for example object interconnectivity varying with phases of a computation. However, because of the runtime cost involved, dynamic compilation is typically limited in scope and driven by local information. Despite this limitation, because specialization is only done for situations that actually have occurred, dynamic compilation can give significant performance benefits without producing huge code size increases.

The three critical problems in dynamic compilation are deciding when to do it, how great a scope to recompile, and how to share the results of dynamic compilation. To a large extent, the answers to these questions depend on both the specific costs and temporal dynamics of the parallel system in question. Thus, they can only be answered via experimentation. However, the Concert approach seeks to do dynamic compilation selectively, focusing on program points flagged by the compiler and instrumented inline to obtain path counts. The cost of recompilation is minimized by recompiling from a template provided by the compiler which is simply adapted to the available runtime information. Determining appropriate code sharing policies and recompilation scopes remain open research questions.

Dynamic compilation can give the benefits of static optimization, producing extremely efficient code. In some cases, dynamically compiled code can be as efficient as that based on complete static information; no inline runtime tests are required (see Figure 8). In other cases, dynamic compilation will produce code with inline tests and specialized execution paths, similar to that produced by speculative compilation, only the basis for specialization is determined at runtime. In all cases, the cost of runtime compilation implies a narrower scope of optimization than in earlier stages. Typically, little more than inlining and peephole optimization is feasible. Further, the real costs of dynamic compilation: detecting opportunities, the runtime cost of compilation, and deciding when

```
...                                     ...
INVOKE-METHOD(B, meth1)                 B→meth1;
...  local computation                  ...  local computation
INVOKE-METHOD(B, meth2)                 B→meth2;
...  local computation                  ...  local computation
INVOKE-METHOD(B, meth3)                 B→meth3;
...                                     ...
```

Figure 8: Dynamic Compilation Example

to invalidate specialized code are not reflected in the code sequences. In Concert, these costs are reduced by using only localized information and using templates to reduce the cost of compilation (specialized for the optimizations of interest). Because of the costs involved, dynamic compilation is likely to only be effective if its cost can be amortized by a long running, typically iterative, computations.

## 3.4 High Performance Runtime System

The fourth tier of the Concert system is a high-performance runtime system which provides efficient runtime operations, exposes the important cost distinctions for optimization by the compiler, and supports the use of speculative and dynamic compilation techniques. Because of the close partnership between the compiler and runtime, the runtime design is an integral part of the Concert system. Below, we describe the cost hierarchy for basic operations on stock hardware multicomputers and outline how the Concert runtime exposes the hierarchy for compiler optimization. System support for speculative and dynamic compilation as well as runtime optimization (scheduling, clustering, load balancing, etc.) and essential runtime services (storage allocation, garbage collection, basic thread scheduling, etc.) is also discussed.

Efficient concurrent object-oriented language implementations must provide a global object namespace, communication services for remote method invocation, and support for scheduling method invocations. Though implementations on custom hardware [15, 36, 42] focus on providing a few general-purpose primitives, runtime systems on stock hardware require a different approach. The hardware structure of such systems necessarily implies a hierarchy of costs for many basic runtime operations. These cost distinctions must be recognized and managed to obtain efficient execution of fine-grained concurrent object-oriented programs.

The Concert approach includes a runtime system design which exposes these critical cost distinctions, so that they can be managed. In particular, the full range of runtime primitives are exposed to the compiler, allowing optimization based on static program analysis. The functionality of the runtime can be divided into three basic parts: a hierarchy of versions of basic runtime operations, support for speculative and dynamic compilation, and fast implementations of essential runtime services.

Basic operations such as communication, invocation, and name service can have dramatically different cost, depending on the generality of the operation required. On stock hardware in particular, these differences can be greater than two orders of magnitude. These differences are illustrated in Table 1 and discussed in detail in [25]. The ability to choose the cheapest appropriate mechanism is essential to achieving good performance. For example, the runtime provides two versions of name translation, one for node-local names, and the other involving global names which may require re-

| *Implementation* | *Clock rate* | *Send/Reply* | | *Invocation* | | *Translation* | |
|---|---|---|---|---|---|---|---|
| | *MHz* | *μs* | *cyc.* | *μs* | *cyc.* | *μs* | *cyc.* |
| CST(on J-m/c) | 28.3 | 6.36/7.8 | 180/220 | 0.6/1.6 | 18/46 | 0.9/2.6 | 26/74 |
| ABCL(on EM-4) | 12.5 | 9 | 112 | 0.24 | 3 | – | – |
| Concert CA(on CM-5) | 33.0 | 0.15/15.7 | 5/520 | 0.12/44.2 | 4/1456 | 0.03/16.1 | 1/536 |

Table 1: Minimum/Maximum Costs of Runtime Operations on Custom and Stock Hardware.

mote access for translation. Incorporating different versions in the runtime interface exposes the cost distinctions, allowing the compiler to choose the appropriate functionality. It is interesting to note that the cheapest versions of operations are as inexpensive as corresponding constructs in traditional sequential languages, while the most general operations are considerably more expensive. The cheapest versions of operations also cost less than the general purpose operations provided on custom hardware.

The Concert runtime also provides support for speculative and runtime compilation. Mechanisms are provided for runtime checking of object properties such as type and location. An assertion maintenance module supports the insertion and invalidation of assertions to support speculative compilation. The runtime guarantees callbacks when assertions are invalidated. Hints can be presented to the system via a set of runtime calls. We are exploring a variety of approaches to exploiting the information provided by hints in the runtime system. The runtime system also provides basic functionality for inline instrumentation and dynamic code caching. These services can be used to manage dynamic compilation and the best form for such services is still an open research question.

The Concert runtime also provides efficient implementations of essential runtime services such as garbage collection, load balancing, and intra-processor scheduling. In particular, sophisticated group scheduling mechanisms, can enhance data locality and state reuse, producing significant performance improvements. These mechanisms may or may not be managed by the compiler. Even without close coupling, they give improvements in overall execution efficiency.

### 3.5 Summary

The Concert approach integrates compiler and runtime efforts to provide optimization at all levels from fully static to fully dynamic. This integration allows program information to be carried forward from phase to phase of the program execution. Thus, the Concert approach conserves critical program information and maximizes the opportunities for optimizing fine-grained concurrent object-oriented programs. As we have discussed, integration requires changes not only to the compiler design, but also to the runtime. In some cases, the runtime yields control to the compiler, in others the runtime provides services which enable aggressive compilation. Finally, efficient execution ultimately rests on the speed of the underlying runtime implementation, so the efficiency of the runtime system is critical.

## 4 Evaluation

We are building the Concert system, an embodiment of the techniques described in this document, to explore and demonstrate the effectiveness of a wide range of program analyses and optimizations for grain size tuning. A thorough evaluation of the Concert approach requires an application program suite, compiler, and runtime system. We are developing an application suite with various application domains, computational structure, and programming style. Based on the application suite, our program analyses and optimizations techniques will be evaluated, tuned, and improved. Experimentation with an entire system is essential since many of the optimizations involve cost tradeoffs, and interactions amongst optimizations must be explored in full scale experiments. The result of our studies will be a thorough evaluation of the effectiveness of a range of program analysis and optimization techniques. Ultimately, these studies contribute to an evaluation of the performance implications of incorporating concurrent, object-oriented features in a programming language.

Some of the specific questions we are exploring include:

- **Static Analysis:** How much precise type information is available from type inference? How much structure information is available from structure analysis?

- **Static Optimization:** How effectively can type and structure information be exploited for locality optimization and what impact does that have on program concurrency? What efficiency gains can be obtained from static binding and object fusion? To what extent does inlining preserve precise type information for optimization? What fraction of the dynamically optimizable opportunities does static optimization capture?

- **Speculative Optimization:** What are the basic tradeoffs in application of the three levels of speculative optimization? How effective are speculative locality checks and how do they interact with placement policy? How effective are assertion-based optimizations, particularly in providing the opportunity for interprocedural optimization? How well can the compiler insert hints to the runtime system? How can the runtime best exploit this information? What is the tradeoff between code size and program efficiency?

- **Runtime Compilation:** How many cost-effective opportunities are there for runtime compilation? What techniques are most effective in identifying these opportunities and exploiting them with minimal cost? What is the right program representation for runtime compilation?

- **Runtime:** What is the frequency of callbacks for assertions? How does this vary under different compiler and runtime policies? How can the runtime exploit additional compiler information (not extant in the program code) at runtime? How should dynamically compiled code be cached? Shared?

- **General:** How effective are all of these techniques for increasing execution grain size? What impact does this have on program concurrency? What is the cost of the object-oriented features in terms of compile and runtime overhead?

It is clear that concurrent, object-oriented languages empower programmers, by allowing them to encapsulate irrelevant detail. However, for COOP languages to become the parallel programming vehicle of choice, serious questions about their efficient implementation must be addressed. In particular, language features in COOP languages appear to require aggressive optimization to achieve acceptable levels of efficiency. The Concert project is developing such aggressive optimization techniques and evaluate them against an application suite of concurrent object-oriented programs. Only with such studies will it be possible to demonstrate convincingly that COOP languages can be made to run efficiently on stock parallel machines. The ultimate goal of such studies is to show that COOP languages are a viable and attractive basis for efficient parallel computation.

## 5 Background and Related Work

The Concert system is related to prior work on efficient implementation of both object-oriented [22, 31, 16] and parallel systems [20, 26]. In particular, many of the optimizations we have discussed have their inspiration in the techniques developed by the SELF compiler group [5]. Of course, the major distinction is that our work focuses on the problems associated with concurrency, distribution, and data parallelism.

Our work also differs in emphasis from a wide variety of work in the area of concurrent object-oriented languages. We are primarily concerned with efficiency, while a variety of projects are concerned primarily with language features [28, 3, 2, 43]. The most closely related work in this area is the ABCL project [42, 37, 43] which is also pursuing efficient implementations. A recently published description of their runtime parallels many of the techniques found in our optimized runtime system. While our research goals are similar, ABCL is significantly different from our source language as it has no support for parallel collections. Further, to date, the ABCL group has focused primarily on runtime techniques and not compiler analysis and optimization.

Recently, a great deal of attention has been focused on concurrent languages based on C++ [35] extensions. ESKit C++ [34], Mentat [18], CHARM++ [24], and Compositional C++ [7] are medium-grained languages in which the programmer supplies grain-size information. These languages integrate concurrency and object-orientation, but requiring the programmer to specify a grain size for efficiency limits program scalability and portability. Typically, the specified grain size is large and limits scalability. Automatic parallelization of large grains is known to be quite difficult. Further, to date, none of them has focused on developing the compiler support necessary to automatically adjust grain size, the primary focus of the Concert system.

The pC++ [27] language supports data parallel operations, but the object-oriented framework of the language allow encapsulation. This means that pC++ programs can express restricted heterogenous concurrency within collections. However, true task level parallelism cannot be expressed. The parallel collections in pC++ are similar to aggregates in Concurrent Aggregates [10]. In the Concert system, data parallelism is expressed as task level concurrency, providing greater programming power, but making efficient implementation significantly more difficult. Effective grain size tuning must be achieved to make data parallel operations efficient.

With respect to parallel systems in general, a wide variety of approaches to portable, programming are being actively pursued. We relate the analogous work on compiling for efficiency in each of these models to grain-size tuning in concurrent object-oriented languages. A subset of other approaches to parallel programming can be loosely classified as data parallel, functional, and committed choice.

Data parallel approaches [38, 19, 9, 4] express parallelism across arrays, collections, or program constructs such as loops in the context of a single control flow model. Data parallel programs admit a degree of grain size tuning, operations in a data parallel operation can be grouped and scheduled together. However, data parallel languages have difficulty expressing task level concurrency or irregular concurrency. Further, all of the data parallel languages provide essentially no support for encapsulation or modularity.

Functional programming approaches [41, 14, 23] have the advantage of determinacy, but have limited expressive power due to the absence of state. If laziness or non-strictness is incorporated, efficient compilation becomes difficult. The particular problem is a similar one of grain size tuning, but under much more difficult circumstances where little synchronization and data reference information may be available at compile time.

Concurrent logic programming approaches, particularly those based on the committed-choice model [40, 17, 33, 32] are similar to concurrent object-oriented languages. However, they have little support for encapsulation and parallel collections. In committed choice languages, the emphasis is on task parallelism which is often expressed as operations on a stream. While some degree of grain size tuning can be achieved grouping successive elements of a stream, the structure of programs essentially limits this to structures analogous to data parallelism.

## 6 Summary and Current Status

We have presented a comprehensive and unified optimization system for concurrent object-oriented languages. Our system includes aggressive static analysis followed by static optimization, speculative optimization and dynamic compilation. All these transformations are leveraged by a high performance runtime system. The key to our approach is exploitation of information at the earliest possible optimization stage and its preservation for use in later stages. Using our system we hope to show that concurrent object-oriented languages can be an efficient medium in which to express a wide variety of parallel computations.

The Concert system has been operational on both sequential and parallel platforms since October 1992. The system includes an optimizing compiler for an extended version of Concurrent Aggregates [11] and a high performance runtime system which runs on both Sun workstations and the Thinking Machines CM5 [25]. We are currently building a second generation system with much greater analysis and optimization capability as well as much greater overall performance. This second generation Concert system consists of a new implementation of the compiler (in progress) and a new implementation of the runtime system (already complete). The compiler, runtime, and the application suite which we are developing for them are detailed below.

We are currently building a second generation compiler which implements the full range of static analysis and program transformation described in this document. This compiler will enable us to experiment with a broad range of optimization approaches. This second generation compiler uses relatively traditional internal data structures based on the program dependence graph (PDG) in Static Single Assignment form (SSA). Novel aspects of the compiler include a constraint-based type inference system and an attributed value system similar to that used in the SELF compiler [5]. We are currently developing the structure analysis and optimizations subsystems.

The second generation runtime system which supports the full range of runtime operations described in this document and in [25] has been operational since March 1993. Novel aspects include: 1) providing a hierarchy of functionality and cost for each runtime operation, allowing selection of the cheapest version and 2) providing support for speculative and dynamic compilation. The complete second generation system (improved compiler and runtime system) should be available for external use sometime during the Fall of 1993.

We are also developing an extensive application suite which will form the basis for optimization experiments. We believe concurrent object-oriented languages are appropriate for a wide variety of numeric and non-numeric applications, giving greatest advantage in problems with irregular computational structure. Our application library will ultimately include regular numeric computations, as a basis for comparison with alternatives such as HP Fortran, irregular numeric applications such as sparse matrix and n-body interaction problems, regular non-numeric applications, and irregular non-numeric applications such as discrete event simulations. Currently, our application suite includes a math library, logic simulator, an n-body interaction solver, particle-in-cell code, and a PC board router. These applications contain thousands of lines of Concurrent Aggregates code and run sequentially on a simulator and in parallel on the CM5.

## Acknowledgements

The Concert project and the authors of this paper are supported in part by grants from the National Science Foundation, grant CCR-9209336, Office of Naval Research, grant N00014-92-J-1961, and the 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] Pierre America. POOL-T: A parallel object-oriented language. In Aki Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199–220. MIT Press, 1987.

[3] W. C. Athas and C. L. Seitz. Cantor User Report Version 2.0. CalTech Internal Report, January 1987.

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

[5] 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.

[6] Rohit Chandra, Anoop Gupta, and John L. Hennessy. Data locality and load balancing in COOL. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993.

[7] 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.

[8] D. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, pages 296–310, June 1990.

[9] Chen and Cowie. Prototyping FORTRAN-90 compilers for massively parallel machines. In *Proceedings of SIGPLAN PLDI*, 1992.

[10] 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.

[11] Andrew A. Chien. *Concurrent Aggregates: Supporting Modularity in Massively-Parallel Programs.* MIT Press, Cambridge, MA, 1993.

[12] Intel Corporation. Paragon XP/S product overview. Product Overview, 1991.

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

[14] D. Culler, A. Sah, K. Schauser, T. von Eicken, and J. 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] W. J. Dally, A. Chien, S. Fiske, W. Horwat, J. Keen, M. Larivee, R. Lethin, P. Nuth, S. Wills, P. Carrick, and G. Fyler. The J-Machine: A fine-grain concurrent computer. In *Information Processing 89, Proceedings of the IFIP Congress*, pages 1147–1153, August 1989.

[16] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Eleventh Symposium on Principles of Programming Languages*, pages 297–302. ACM, 1984.

[17] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice-Hall, 1990.

[18] A. Grimshaw. Easy-to-use object-oriented parallel processing with Mentat. Overview for Mentat system, 1992.

[19] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler Optimizations for FORTRAN D on MIMD Distributed-Memory Machines. In *Supercomputing '91*, pages 86–100, November 1991.

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

[21] 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.

[22] R. E. Johnson, J. O. Graver, and L. W. Zurawski. Ts: An optimizing compiler for smalltalk. In *OOPSLA '88 Proceedings*, pages 18–26, September 1988.

[23] Simon L Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

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

[25] Vijay Karamcheti and Andrew Chien. Concert – efficient runtime support for concurrent object-oriented programming languages on stock hardware. Submitted to SUPERCOMPUTING'93.

[26] D. J. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM Symposium on Principles of Programming Languages*, pages 207–18, 1981.

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

[28] Carl R. Manning. Acore: The design of a core actor language and its compiler. Master's thesis, Massachusetts Institute of Technology, August 1987.

[29] N. Oxhøj, J. Palsberg, and M. Schwartzbach. Making type inference practical. In *Proceedings of OOPSLA '92*, 1992.

[30] John Plevyak, Vijay Karamcheti, and Andrew Chien. Analysis of dynamic structures for efficient parallel execution. Revised Paper, submitted to LCPM '93.

[31] A. D. Samples, D. Ungar, and P. Hilfinger. Soar: Smalltalk without bytecodes. In *OOPSLA '86 Prodeedings*, pages 107–18, September 1986.

[32] V. Saraswat. *Concurrent Constraint Programming Languages.* MIT Press, 1992. To appear, also available as Technical Report from Carnegie-Mellon University as Technical Report CMU-CS-89-108.

[33] 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.

[34] 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.

[35] Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley, second edition, 1991.

[36] T. Baba, et al. A parallel object-oriented total architecture: A-NET. In *Proceedings of IEEE Supercomputing '90*, pages 276–285. IEEE Computer Society, 1990.

[37] K. Taura, S. Matsuoka, and A. Yonezawa. An efficient implementation scheme of concurrent object-oriented languages on stock multicomputers. In *Proceedings of the Fifth ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, 1993.

[38] Thinking Machines Corporation. *Getting Started in CM Fortran*, 1990.

[39] Thinking Machines Corporation, Cambridge, Massachusets. *CM5 Technical Summary*, October 1991.

[40] K. Ueda and M Morita. A new implementation technique for flat GHC. In *Proceedings Seventh International Conference on Logic Programming*, pages 3–17. MIT Press, 1990. Revised version to appear in New Generation Computing.

[41] Yale University, New Haven, Connecticut. *Report on the Programming Language Haskell*, 1.0 edition, April 1990.

[42] M. Yasugi, S. Matsuoka, and A. Yonezawa. ABCL/onEM-4: A new software/hardware architecture for object-oriented concurrent computing on an extended dataflow supercomputer. In *Proceedings of the ACM Conference on Supercomputing '92*, 1992.

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

## Contents

**1 Introduction** **1**

**2 Basic Language Model** **4**

**3 Concert Approach** **5**

3.1 Static Analysis and Optimization . . . . . . . . . . 7

3.1.1 Type Inference . . . . . . . . . . 7

3.1.2 Structure Analysis . . . . . . . . . . 8

3.1.3 Exploiting Static Information . . . . . . . . . . 9

3.2 Speculative Transformation and Runtime Support . . . . . . . . . . 10

3.3 Dynamic Compilation and Specialization . . . . . . . . . . 13

3.4 High Performance Runtime System . . . . . . . . . . 14

3.5 Summary . . . . . . . . . . 15

**4 Evaluation** **15**

**5 Background and Related Work** **17**

**6 Summary and Current Status** **18**