The ICC++ Programmers' Reference

A. Chien, J. Dolby, V. Karamcheti, J. Plevyak and X. Zhang

July 30, 1995

Introduction

This document describes the design of ICC++, a new parallel C++ dialect. ICC++ is designed to support both concurrent and sequential programming, and the design priority is to minimize differences with C++ while providing a flexible parallel model. ICC++ is a restricted superset of C++: some of the more troublesome constructs of C++ are reigned in, and the language is extended for expressing parallelism.

- ICC++ generally preserves the basic syntactic structure of C++, permitting ICC++ programs to be translated to C++ with modest effort. This allows a single source to be maintained for parallel and sequential programs, leveraging the programming environments that exist for C++.

This document starts with an overview of the sequential language, focusing on its essential similarity to C++ and describing the how its differences effect sequential programming. The remainder of the document covers ICC++ language extensions. First, we explain the extensions to the C++ object model required to support concurrent programming, mainly object-based concurrency control, and then the explicitly concurrent statements are described: concurrent blocks and concurrent loops. We next detail collections, which both integrate arrays into the C++ object model and extend them for parallel programming. The last major feature of ICC++ is a set of annotations involving locality and aliasing designed to facilitate compiler optimizations. Finally, we cover linking to external libraries (such as existing C++ code) and conclude with a succinct grammatical summary of ICC++ changes to C++.

This document is a programmers reference for the ICC++ language, in which comprehensiveness is more important than brevity. For a concise definition of the language, see [2]. The motivation of the language design is discussed in [1].

Conventions

In some places both C++ and ICC++ code are used side-by-side for comparison. Since the codes look similar, they are distinguished by using `this font for C++` and **`this font for ICC++`**.

Contents

List of Figures

1 Sequential Features

ICC++ is closely derived from C++, and retains most of the language intact. Classes, objects, members and other basic features are intact; arrays have been extended significantly, but their usage remains the same if the extensions are not used. The five major changes are the following. Firstly, ICC++ integrates arrays into the object model; this allows user-defined array classes with all of the attributes of standard ones. These user-defined array classes, called *collections*, are described in Section 4. Secondly, pointers to built in types are forbidden. Thirdly, the semantics of casts has been modified so that all casts are either statically safe or are checked at runtime. Fourthly, unions have been forbidden as they, like unsafe casts, vitiate type safety. Finally, the `extern` form has been generalized to accept C++ as well as C declarations.

1.1 Pointers

There are three ICC++ features that affect pointers: integrating arrays into the object model, forbidding pointers to non-object types and automatic dereferencing.

1.1.1 Pointers and Arrays

Arrays in ICC++ are first-class objects; they are defined by class definitions and have all the attributes of standard classes. These arrays have a new syntax, based upon the `element_type [ ]` type specifier: all declarations of arrays must use the `element_type array_name[]` syntax. In essence the syntax requires programmers to indicate when arrays are being used[1], using familiar C++ style declarations. The `element_type[ ]` syntax in ICC++ is allowed everywhere and declarations of the form `int foo[ ][ ]` are permitted. This new use of `element_type[ ]` syntax is shown in Figure 1 in the ICC++ code fragment on the right. Note the declaration of the function and of `new_array`; both of these show how the new syntax must be used. This breaks the connection between pointers and arrays, and so inter-conversion between them is forbidden. Thus, the standard practice of using pointer declarations for arrays, as shown in the left hand of Figure 1 is forbidden.

- Divorcing pointers and arrays, and integrating arrays into the object model is increasingly common in C++ dialects such as [3, 7] because it can improve both language safety and analyzability.
- Note that separating pointers and arrays makes pointer arithmetic useless, since it was only defined within arrays (see [4]). Hence, pointer arithmetic is forbidden in ICC++

```
int *copy(int sz, int *stuff) {       int copy(int sz, int stuff[])[] {
  int *dup = new int[sz];               int dup[] = new int[sz];
  for(int i = 0; i < sz; i++)           for(int i = 0; i < sz; i++)
    dup[i] = stuff[i];                    dup[i] = stuff[i];
  return dup;                           return dup
}                                     }
```

Figure 1: ICC++ changes to C++ array syntax

[1]In C and C++, pointers can be used to refer to both individual objects as well as arrays of objects.

1.1.2 Pointers to Non-Objects

The other restriction of pointers is more fundamental; pointers to built in types are forbidden. No `int *` or `char *` pointers are allowed in ICC++. Some such pointers are used to represent arrays, in which case they can simply use the `type [ ]` syntax; however, other uses of such pointers will need adaptation. A common use of such pointers is to "return" multiple values from a function call; this can be expressed using tuples, as described in Section 1.3. Furthermore, the lack of pointers to built in types can always be worked around by creating an object with one field of the desired type and taking a pointer to that. This is illustrated in Figure 2; the ICC++ code is on the right.

- The lack of pointers to built in types permits more thoroughgoing storage of these type in registers, as there can be no aliasing either within or between concurrent computations.

```
int sum(int l, int *primes) {
 int sum;
 for(int i = 0; i < l; i++) {
   sum += i;
   if (is_prime(i))
     (*primes)++;
 }
 return sum;
}
```

```
struct Int {
 int val;
}

int sum(int l, Int *primes) {
 int sum;
 for(int i = 0; i < l; i++) {
   sum += i;
   if (is_prime(i))
     primes->val++;
 }
 return sum;
}
```

Figure 2: Faking pointers to built in types in ICC++

1.1.3 Automatic Dereferencing

Traditional object-oriented languages provide *object names*. Pointers are the closest thing that C++ has to such names, but they make an explicit distinction between the pointer and the object which is absent for object names. To allow pointers to be used as object names, ICC++ blurs this distinction by supporting *implicit dereferencing* of object pointers used in member function calls: when an operator or function is called on a pointer, that function is applied to the referent.

The only exceptions to this rule are `operator =` and `operator *` which are both defined for pointer types. No other operators are defined for pointers since ICC++ forbids pointer arithmetic.

This is shown in Figure 3. In C++, `t += v` would add the two pointer values, but in ICC++, implicit dereferencing calls `foo::operator +=(foo *)`. The other two operations illustrate the cases where implicit dereferencing is not applied: assignment and dereferencing operations.

1.2 Casts

All casts in ICC++ are required to be safe, and are checked at runtime if necessary. All casts that cannot be checked are forbidden. Essentially, this forbids inter-conversion of unrelated pointer types and casting between an arithmetic type and a pointer type. Furthermore, all casts down

```
class foo {
  int a;

 public:
  foo *operator +=(foo *i)  a += i.a; return this;
  foo *operator =(foo *i)  a = i.a; return this;
  int operator *(void)  return a;
}

void main(void) {
  int a;
  foo *t = new foo;
  foo *v = new foo;
  t += v;              // auto-dereference and foo::operator +=

  t = v;               // assignment is legal on pointers: no auto-dereference

  a = *t;              // no auto-dereference as * is legal for pointers,
}                     // so this would be a type error, as *t is no int
```

Figure 3: Automatic dereferencing of pointers in ICC++

inheritance hierarchies are done at runtime using runtime type information. Thus, they correspond to `dynamic_cast` rather than `static_cast` in the new C++ casting terminology. As with `dynamic_cast` all failing casts return `NULL`. All other casts are exactly as in C++.

[6, 4] define a new cast syntax consisting of four cast operators: `static_cast`, `dynamic_cast`, `const_cast` and `reinterpret_cast`. These operators are modified as follows:

static_cast is only allowed for coercions that may be done implicitly. It cannot be used to navigate down inheritance hierarchies (its main purpose) because it does not do runtime checks.

dynamic_cast is permitted. Inheritance navigation done by `static_cast` in C++ must use `dynamic_cast` in ICC++ because it checks at runtime to ensure the conversion is correct.

reinterpret_cast is forbidden.

const_cast is allowed without restriction.

- These restrictions upon casting enable ICC++ to provide garbage collection. Casting pointers into arithmetic types greatly reduces the effectiveness of garbage collection, and even unsafe casts can do so by confusing the system about what pointers it actually has.
- Unsafe casts are generally regarded as bad style in application-level programming. While they can be necessary in low-level system code such as device drivers, ICC++ is not intended for that level of programming.

Because all casts are statically safe or checked at runtime programmers can write type-safe code by simply testing for NULL return values from any `dynamic_cast`s. Because ICC++ can check types at runtime, programmers can safely downcast in many cases that would be prohibited in C++. As an aid to finding logical programming errors, code can have diagnostics added to print useful errors when these casts fail; this is illustrated in Figure 4.

```
struct foo {
 int a;
 foo *buddy;
};

struct bar : public foo {
 int b;
 void do_work(void);
};

void do(bar *me) {
 bar *bud = (bar *) me- >buddy;
 bud- >do_work()
}
```

```
struct foo {
 int a;
 foo *buddy;
}

struct bar : public foo {
 int b;
 void do_work(void);
}

void do(bar *me) {
 bar *bud = (bar *) me- >buddy;
 if (bud == NULL)
   cout << "do: bad buddy found";
 else
   bud- >do_work()
}
```

Figure 4: Handling downcasts in ICC++

Figure 4 illustrates the advantage of checking casts at run time. As with the `dynamic_cast` operator, casts that fail return the null pointer; this allows the user code to handle failure explicitly.

1.3 Tuples

ICC++ provides *tuples* to allow multiple return values for functions. A tuple is essentially an implicit `struct` in which fields can be accessed only by destructuring, and are declared using the syntax `( <type>, <type>, ... )`. Tuples can only be returned by functions and can only be used in assignment; as suggested if Section 1.1.2, tuples can be used in place of pointers to return multiple values. This is illustrated in Figure 5.

- Using tuples rather than pointers to return multiple values allows the return values to passed directly through the stack or as one communication operation in a parallel machine.

1.4 Unions

The last restriction imposed by ICC++ is that `union`s are not allowed. Unions create a loophole which reduces the runtime type safety of programs, and they are already emasculated in C++ since they cannot contain fields of types that have constructors nor can they be used for derivation. Derivation can be used to represent unions (see Figure 6), at some cost in storage.

1.5 `extern`

Just as C++ permits access to C code using `extern` declarations, so ICC++ allows access to both C++ and C code. ICC++ extends the `extern` declaration slightly to allow the form `extern "C++"` *declaration*. The `extern` declaration indicates that the form is implemented in another language and all appropriate format conversions and calling conventions will be automatically handled. The `extern "C"` works just as it does in C++, but `extern "C++"` is more involved, as Figure 7 suggests.

```
double
my_div(double a,
        double b,
        double *q)
{
  q = a % b;
  return a / b;
}

void main(void) {
  int foo, bar;
  foo = my_div(88.7, 5, &bar);
}
```

```
(double, double)
my_div(double a, double b) {
  int quot = a / b;
  int rem = a % b;
  return(a,b);
}

void main(void) {
  int foo, bar;
  (foo, bar) = my_div(88.7, 5);
}
```

Figure 5: Using multiple return values in ICC++

```
union foo {
 int a;
 float b;
};

void dump_int(foo *a) {
  cout << a->a;
}
```

```
struct foo {};
struct foo_int : public foo {
 int a;
};
struct foo_float : public foo {
 float b;
};

void dump_int(foo *a) {
  foo_int b = (foo_int *) a;
  cout << b->a;
}
```

Figure 6: Faking unions via derivation in ICC++

In Figure 7, the class `foo` can be used normally in ICC++ although it is a C++ class. Any differences in the object format or calling conventions between the ICC++ implementation and the C++ implementation will be handled by the ICC++ system. Just as the C++ extern mechanism is not designed to allow C++ objects to be passed into C[2], so the ICC++ extern mechanism does not generally allow ICC++ objects to be passed into C++ or C. For this, a more general interoperability scheme is required, such as the IDL interface defined in a forthcoming Appendix to this manual.

1.6 Summary

ICC++ makes several significant changes to C++, most prominently reworking the syntax for arrays and forbidding pointers to built in types. Changes are also made to casting and unions are prohibited. The change to arrays increases the static type safety of programs by preventing confusion between arrays and pointers to single objects; type safety is further enhanced by banning unions and prohibiting unsafe casts. Disallowing pointers to built in types allows implementations

[2]C++ class declarations will not parse in any C compiler if they use C++ constructs such as member functions.

```
extern "C" {
#include <stdio.h>
}

extern "C++" {
 class foo {
   int a;

  public:
   int get_a(void);
   int set_a(int);
 };
}

void main(void) {
 foo *a = new foo;
 conc {
  a->set_a(5);
  printf("set_a called with 5.");
 }
}
```

Figure 7: ICC++ code using `extern "C++"`

complete freedom in caching them, without fear of aliasing issues. ICC++ also provides some extensions to the C++ syntax, allowing multiple values to be returned conveniently from functions and making pointers behave more like traditional object names. In the following sections, we discuss extensions to C++ to support high performance and concurrency.

2 Concurrent Objects

The core of object-oriented programming is building abstractions – encapsulated data and program which define a well-specified interface. The abstraction model utilizes a set of accessor methods that perform logically atomic operations upon the abstraction's state; each operation must maintain the consistency of that state. Concurrency allows only a partial order on state updates, complicating the notion of consistency. Any concurrent model must preserve the notion of logically atomic operations upon an abstraction in a concurrent setting.

To support concurrent abstractions, ICC++ extends C++ by introducing explicit object-based concurrency; this requires a significant extension to the object model with respect to objects in C++: a concurrency control scheme is introduced to ensure consistency of object state, which provides a framework for reasoning about the behavior of concurrent programs. ICC++ provides both concurrency control and concurrency guarantees which specify respectively at most and at least how much concurrency a program will have.

2.1 Object Consistency

The notion behind object-based concurrency control is that objects assure the consistency of their own state. Concurrent calls upon the object interface are not allowed to interfere with each other, meaning that calls can only run concurrently if such execution is equivalent to some locally sequential order of the calls. Effectively, two calls can execute concurrently only if neither methods writes any member the other one reads.

This notion of consistency applies only to the actual instance variables themselves; that is to the actual state of the object. Thus for a member of type `foo *` sequentializability is enforced only for the pointer itself, not for the object to which the pointer refers. Similarly, any state that is not part of the object is not protected, so updates to global variables from within methods will not have any concurrency control. For this reason, static member are *not* considered part of any object for concurrency control purposes.

- It might seem that static members should be part of every object of that class; however, this would entail one of two problems. Either there would still be race conditions amongst updates from different objects or global serialization would be needed across all objects of that class to prevent such races.

2.1.1 Member Functions

Some examples show how this rule applies to member functions. Figure 2.1.1 illustrates simple sharing patterns: members `left(void)` and `right(void)` can be concurrent as can multiple calls to one of them. Such calls both read the same data, but do not write and so cannot interfere. Concurrent calls to `set_right` and `set_left` are also permitted, as neither writes anything the other might read. However, calls to `left` and `set_left` cannot be concurrent, as they can interfere through `left`.

Figure 2.1.1 provides a more complex example. Although `set_length` and `set_height` both read and write the object, neither writes a field the other reads, and so they can proceed concurrently. A hairier case is multiple calls to `set_length`; while neither writes a field the other reads, multiple interleaved methods writing the same state may not be sequentializable[3], and so these calls would not be concurrent.

[3]At least, not if it writes more than one field.

```
class Line {
  int _left;
  int _right;

 public:
  int left(void) { return _left; }
  int set_left(int i) { return _left = i; }
  int right(void) { return _right; }
  int set_right(int i) { return _right = i; }
};
```

Figure 8: Member functions requiring simple concurrency control

```
class Region {
  int _top;
  int _left;
  int _right;
  int _bottom;

  int _length;
  int _height;

 public:
  int set_length(void) { return _length = _right - _left; }
  int set_height(void) { return _height = _top - _bottom; }

  int area(void) { return  _length * _height; }
};
```

Figure 9: Member functions that must be sequentialized

2.2 Consistency across Objects

ICC++ ensures the consistency of single objects, but sometimes consistency is desired across multiple objects. ICC++ provides two mechanisms for this: the `integral` declaration and `friend` functions.

2.2.1 `integral`

The `integral` declaration is a type specifier used like `const` that can be applied only to member variables. It extends the concurrency control semantics of the object to include that member. Thus, two methods can run concurrently on an object only if such execution is equivalent to some sequential order for both that object and all members declared `integral`. For instance, the `buckets` member in Figure 2.2.1 must be declared `integral` to incorporate it into the hash table's concurrency control. If it were not, then calls to `add` could run concurrently since it does not effect the state of the `HashTable` itself. If this were to happen, interleaving calls to `find` and `add` could cause the same element to be inserted multiple times. However, since `find` reads and `add` writes the state of `buckets`, declaring `buckets` to be `integral` prevents this from happening.

- Note that this is no guarantee that other member functions on an `integral` object will not interfere with `add` or `find`. Other member functions could be called upon the array named by `buckets` via some other pointer to it. The `integral` declaration applies the `HashTable`'s concurrency control to the `buckets` member. *The only guarantee is that calls from this object using the name* `buckets` *will be sequentializable.*

```
class HashTable {
  integral Bucket *[] buckets;
  int n_buckets;

  Bucket *find_bucket(Key k) {
    return buckets[k.hash%n_buckets];
  }

 public:

  Element find(Key k) {
    return find_bucket(k)->find(k);
  }

  Element add(Element e) {
    Bucket *b = find_bucket(e.key);
    if (!b->find(e.key)) b->add(e);
    return e;
  }
};
```

Figure 10: Extending concurrency control to contained objects using `integral`

2.2.2 `friends`

`friend` functions in C++ are considered member functions upon all `friend`ly arguments, and thus `friend` functions in ICC++ can be used to procedurally compose operations on several objects into a single consistent operation subject to the same object consistency and concurrency guarantees as above. That is, the `friend` function will be consistent with respect to all of the objects for which it operates as a friend.

Consider an example of `friends` (see Figure 2.2.2) in which a matrix is multiplied by a vector. Neither argument may be changed while the `operator *` is running. Declaring `operator *` to be a `friend` of both the matrix and vector classes produces this effect by incorporating it into the concurrency control of both classes.

2.3 Concurrency Guarantees

The other aspect of concurrent semantics is concurrency guarantees. Essentially, two members are guaranteed to run concurrently if they obviously need not be sequentialized. That is, the member code and any nested calls on `this` are examined for explicit accesses to object state. Two members *must* run concurrently if neither one can possibly write any object state that might be read or written by the other. Figure 2.3 shows a simple example of methods with guaranteed

```
class Vector;

class Matrix {
  // other state here

  friend Vector operator*(Matrix&, Vector&);
};

class Vector {
  // other state here

  friend Vector operator*(Matrix&, Vector&);
};
```

Figure 11: Extending concurrency control to multiple objects using `friends`

parallelism based upon a relaxation method; the only concurrency guarantee for this class is that `read`s will be concurrent with other `read`s and with `update`. Since `update` and `do_step` both access `running_total`, no concurrency is guaranteed between them.

```
class GridCell {
  int total;
  int running_total;

 public:

  int read(void) { return total; }

  void update(int i) { running_total += i; }

  void do_step(void) {
    total = running_total;
  }
};
```

Figure 12: ICC++ ensures concurrency amongst `reads`

2.3.1 Data Dependence

Figure 2.3 illustrates the common case where it is apparent what state a member accesses; however, this is not always the case. Conditionals, member pointers, indirect function calls and other constructs make the state a method accesses dependent upon the particular invocation. Thus all reads and writes either *will* happen or *may* happen during a particular invocation, as determined be trivial syntactic examination. Concurrency between two methods is guaranteed only when neither method *may* write any state the other *may* read. In Figure 2.3.1, there is in fact no race condition between `update_when_even` and `update_when_odd` because `guard%2` has to be either 1 or 0. However, there is no concurrency guarantee because they both read and write `total`.

- This strict insistence upon simple syntactic analysis that does not even consider obvious cases like simple conditionals is vital. Otherwise, compiler analysis would be required to implement a language guarantee, making language semantics dependent upon current compiler capability and potentially subjecting them to change in step with compiler technology.

```
class Bizarre {
  int total;
  int guard;

 public:

  void update_when_even(int i) {
    if (guard%2==0)
      total += i;
  }

  void update_when_odd(int i) {
    if (guard%2==1)
       total += i;
  }
};
```

Figure 13: Concurrency not guaranteed

2.4 Summary

The concurrency control semantics of ICC++ are based upon maintaining consistent state for objects. Concurrency upon an object is allowed only when methods executing simultaneously when such execution is equivalent to some sequential execution. To make reasoning about deadlock freedom possible, ICC++ also provides a guarantee that concurrency upon a single object will be exploited when it is syntactically obvious that a set of methods cannot effect each others' operation.

- An implementation has plenty of freedom between the concurrency control and guarantee semantics. For instance, the compiler could choose to run the methods of Figure 2.3.1 concurrently, if it could determine that there is no race condition. Even if it could not figure this out, it could still make them concurrent, putting an explicit lock around the `total += i` in each method.

3 Concurrent Statements

ICC++ allows the programmer to explicitly insert concurrency into a program; essentially concurrent statements specify a set of statements to execute and under-specify their order. Block structured concurrency is added by means of concurrent blocks and concurrent loops. ICC++ also provides statements to introduce arbitrary, unstructured concurrency.

3.1 `conc` Blocks

The basic mechanism for introducing concurrency in ICC++ is the `conc` block. A `conc` block is a compound statement, prefixed with the keyword `conc`. This block, an example of which is Figure 14, defines a partial order on its constituent statements. Any pair of statements in a `conc` block can execute concurrently unless an identifier appearing in both is assigned in at least one of them, or the former statement contains a jump statement (`goto`, `break` or `continue`) that may prevent the latter being executed. Blocks are considered ordinary statements for this purpose. For instance, in Figure 14, statements 1 and 2 can execute concurrently, then statements 3 and 4 can execute concurrently. Statement 5 must wait until 4 completes, because 4 contains a `break`.

```
conc {
  double foo = pow(3.0,8);      // 1
  double bar = log(46.7);       // 2
  double baz = foo + bar;       // 3
  if (bar < 0) break;           // 4
  int fuzz = foo;               // 5
}
```

Figure 14: A `conc` block

These rules are designed to expose concurrency upon objects, while preserving sequential semantics where it is natural. This enables the introduction of concurrency with small perturbation to program structure. Sequentializing for local variables allows preexisting compound statements that declare and use local variables to be transformed into conc blocks, exposing concurrency for calls upon objects within them. Similarly, permitting control flow within conc blocks, and providing a natural semantics for it, allows `conc` to be applied to preexisting code where such irregular control structures are used. Indeed, if the concurrency control on objects is sufficient to maintain program correctness, a `conc` block may be introduced without changing program behavior.

Assignment is a decidedly convoluted concept in C++, meaning different things for different types. For arithmetic types such as `int` and `double`, assignment means `operator =` and the myriad update operators such as `operator +=`. For pointer types, ICC++ supports only `operator =`. For object types, the concept of assignment breaks down entirely, as the update operators are user defined methods and even `operator =` itself can be overloaded. In this context, only `operator =` is considered an assignment to an object type, because it "looks like" assignment and even behaves like it when not overloaded.

The results of this are illustrated in Figure 15. In this case, statements 3, 4, 5 and 6 must wait for statement 1 to finish and statement 5 must also wait for statement 2. They can then all execute concurrently, since statement 5 does not assign to `a`, but instead to `a`'s referent. Statements 7 and 8 may start once statement 2 has done; statement 9 waits for them because it assigns `b`. Statement

10 starts once 9 finishes. Finally, ~Foo must be called on b; the destructor is considered to assign b, and starts after statement 10 completes.

```
conc {
  Foo *a = new Foo(5);  // statement 1
  Foo b(5);             // statement 2

  a– >play("happy");     // statement 3
  a– >work("sad");       // statement 4

  *a = b;               // statement 5
                        // (this is the same as a– >operator=(b);)

  a– >play("ecstatic");  // statement 6

  b.work("miserable");  // statement 7
  b.play("joyous");     // statement 8

  b = *a;               // statement 9

  b.work("depressed");  // statement 10
}
```

Figure 15: An involved conc block

The basic semantics of jump statements are preserved in conc blocks; statements will not execute if control flow jumps around them and can execute multiple times if control flow jumps back to them. This is expressed as a dependence between each statement containing a jump and all those statement that it may prevent from execution. The simplest case is a jump, such as break or continue, that exits the conc block. Such jumps create a dependence upon all subsequent statements in the conc block, which cannot execute until the statement containing the jump has completed without jumping. Figure 3.1 has an example of this. Note that this preserves the behavior of break and continue when used in a conc block within a loop or switch.

Jumps within a conc block come in two flavors: forward and backward. A forward jump creates a dependence between itself and all statements in the block between it and its corresponding label. This is illustrated in Figure 16, where the statements marked with a * are dependent upon statement. A backward jump creates dependencies between itself and the next "iteration" of the implicit loop created by the goto. This is shown in Figure 17.

3.2 Concurrent Loops

Each of the C++ looping constructs can be modified by conc producing conc for, conc while, and conc do while. C++ is unusual in that no loop construct has a distinguished loop variable, as does for in Pascal and do in Fortran. Thus, the semantics of the concurrent loop forms must be designed carefully to expose cross-iteration concurrency while retaining reasonable behavior for the local variables. Furthermore, the concurrent loops must be compatible extensions. Since all C++ loops allow control flow operations, the concurrent loops must support them as well. The resulting

```
conc {
  a;
  if (b) goto l1;
  c;                       *
  l1: d;
}
```

Figure 16: forward

```
conc {
  a;
  l1: c;                   *
  if (b) goto l1;
  ...
  d;
}
```

Figure 17: backward

Figure 18: Dependencies caused by unstructured control flow in conc blocks

semantics is that, in a conc loop, loop carried (read after write) dependences are respected only for scalar variables, but not for others such as array dependences and those through pointer structures. Essentially, conc loops are dynamically unfolding conc blocks, with local variables renamed for each iteration, as shown in Figure 19

```
conc while (i < 5) {                if (i < 5)
  a->foo(i);                           conc {
  i += 1;                                a->foo(i);
}                                        i0 = i+1;
                          ==>            if (i0 < 5)
                                          conc {
                                            a->foo(i0);
                                            i1 = i0 + 1;
                                            ...
```

Figure 19: A concurrent loop as a dynamically unfolding conc block

The motivation of this design parallels that of conc blocks. Permitting control flow and respecting scalar variable dependences within concurrent loops simplifies adding concurrency to preexisting sequential loops. As with conc blocks, concurrent loops specify *available* concurrency and make no guarantees about actual concurrency. This allows the implementation considerable latitude in scheduling iterations, such as running groups of iterations sequentially on different nodes.

Figure 20. In this loop, the cross-iteration dependency for i causes the loop counter to increment without race conditions, but there are no restrictions upon the concurrency of the workers[i].do_work() calls.

```
Worker workers[10]
conc for (int i = 0; i < 10; i++)
  workers[i].do_work();
```

Figure 20: A data-parallel loop

A more complex example (Figure 21) illustrates the concurrency control in parallel loops in more detail. The basic idea is that the workers do some work and then dump a checkpoint, repeating until all work is done. Since there is a dependence between the `conc while` loop test and the work portion, the work portions of each iteration will be sequentialized, and the `conc for` inner loops will be sequential as well because they all may write `s`. The checkpoint portion of the `while` iterations will happen concurrently, despite the `continue` statement. The `continue` statement creates a dependence between it and the rest of the body, but once it executes, the rest of the loop *and subsequent iterations* can proceed concurrently. Note that the checkpointing must happen after the computation, because it uses `stage` which the computation updates. The next loop test and iteration can start before the checkpointing has finished because there is no dependence[4]. Thus, this is a partially parallel loop.

```
Worker workers[10];
conc while (stage < last_stage) {
  // do some work
  int s = last_stage;
  bool no_dump = false;
  conc for(int i = 0; i < 10; i++) {
    int p = workers[i].do_stages();
    if (p < 0) no_dump = true;
    else if (p < s) s = p;
  }
  stage += s;
  if (no_dump) continue;

  // dump checkpoint
  for(int j = 0; j < 10; j++)
    workers[j].dump_state(stage);
}
```

Figure 21: A partially parallel loop

The motivation of this design parallels that of `conc` blocks. Permitting control flow and respecting scalar variable dependences within concurrent loops simplifies adding concurrency to preexisting sequential loops. Observe how a fairly complicated loop like Figure 21 could have concurrency exposed by inserting `conc`, while preserving the sequential semantics required by scalar variables like `stage` and the `continue` statement.

3.2.1 Reductions

The semantics of `conc` blocks and hence of concurrent loops provides that a sequence of updates to a simple variable be sequentialized. However, this is not always desirable. When the operations are associative and transitive, they can be more efficiently implemented as parallel reductions. ICC++ provides support for this using the update operators of C; the following operators can be used in reductions: `+=, -=, *=, <<=, >>=`. These operators can be reduced both for arithmetic types

[4]It is up to the `Worker` objects to make sure that `do_stages` and `dump_state` do not interfere with each other.

and user defined types. *It is up to the programmer to ensure that user-defined versions of these operators are indeed associative and transitive if they are used in parallel loops.*

```
extern int a[];
extern int a_size;

// a potential reduction
int total;
conc for(int i = 0; i < a_size; a++)
  if (i%2)
    total += a[i]

// cannot reduce
conc for(int j = 0; j < a_size; j++)
  printf("%d
n", total += a[i]);
```

Figure 22: Reduction examples

As with concurrent loops themselves, there is no guarantee of concurrency for reductions; these operators *may* execute in parallel but are not required to do so. Furthermore, reductions will only be used when the intermediate results are never used.

3.3 Unstructured Concurrency

Concurrent blocks and loops provide a structured mechanism for expressing concurrency within the traditional C control structures; however, sometimes a less structured mechanism is required to express complex concurrency. ICC++ provides `spawn` and `reply` to support this unstructured concurrency. The `spawn` statement generates parallelism and the `reply` function gives the user precise control of caller/callee synchronization.

3.3.1 Spawn

The statement `spawn s;` creates a new thread to execute the statement `s`, which can be an arbitrary statement, including a compound one. All local variables in scope at the `spawn` statement become read-only in the spawned thread, preventing unsynchronized access to them by the spawning and spawned threads.

The spawned and spawning threads are guaranteed to run concurrently, unlike `conc` which is merely a hint. This provides the programmer with more direct control over concurrency, but this guarantee can be expensive to enforce, and generally should be used sparingly.

- In this context, “concurrently” means only that neither *must* wait for the other. The execution of spawner and spawned *will* be interleaved if that is required. This is formally known as *weak fairness.*

3.3.2 Reply

An object `reply` is created for each function call in the program execution; it accepts `operator()` and it has the prototype `void reply(`τ`)`, where τ is the return type of the callee. When called, it returns a value to the caller, just as the `return` statement does, but `reply` does not terminate execution of the caller, allowing caller and callee to run in parallel.

Furthermore, the `reply` object can be passed out of a function, effectively delegating responsibility to returning a value to some other function. Passing `reply` out can be used to implement tail forwarding as well as user defined synchronization structures.

3.3.3 Unstructured Idioms

The `spawn` and `reply` mechanisms can be used to implement customized communication and synchronization structures. A couple of common examples, tail forwarding [5] and barriers, are described below.

Tail Forwarding can accomplished simply using the form `spawn reply(e)`. The expression `e` will be spawned and evaluated in parallel, and then the result will be returned to the caller of the spawning function.

Barriers can be implemented using a user-defined barrier class. Such a class, shown in Figure 23, captures the `reply`s of all the synchronizing functions, and then calls them all at once when everybody has made it to the barrier.

```
class Barrier {
  typedef void (*reply_obj)(...);
  reply_obj replies[];
  int count;
  int index;

 public:
  Barrier(int i) {
    count = i;
    replies = new reply_obj[count];
  }

  wait(void) {
    replies[index++] = reply;
    if (index == count)
      for(int i = 0; i < count; i++)
        (*replies[i])();
  }
};
```

Figure 23: A user-level barrier

3.4 Summary

The concurrent constructs of ICC++ are designed make expressing concurrency natural within a C++ framework. The `conc` blocks allows concurrency to be introduced into ICC++ programs while preserving the familiar sequential semantics for variables which cannot handle concurrency. Concurrent loops permit regular parallelism to be expressed simply, and the unstructured constructs allow customized parallel structures to be used when necessary.

4 Collections

ICC++ replaces C++ arrays with *collections*, which integrate arrays into the object model. A collection consists of an indexable set of element objects and a separate collection state. Both the collection itself and its elements have a class type, which allows both to be treated as objects, permitting members and derivation. Furthermore, this collection of objects provides a convenient form for expressing parallelism and data distribution in a concurrent object-oriented context. Unlike array elements, the constituent elements are aware they are part of a collection, allowing collections to implement an aggregate behavior and interface. A simple collection definition is shown in Figure 24.

```
class Counter[] {
  int elt_total;
  int Counter[]::total;
 public:
  Counter(void);
  Counter[](void);

  int count(int);
  int elt_sum(int);
  int Counter[]::sum(void);
};
```

Figure 24: An ICC++ collection

This declaration creates two classes: the `Counter[]` collection type and the `Counter` element type. The `Counter` element has just one field: `elt_total`. The `Counter[]` collection consists of a linearly addressable set of `Counter` objects and one field of collection state: `total`. `Counter[]` has the member function `sum`, and `Counter` has the member functions `count` and `elt_sum`. Notice that the collection declarations are qualified and the element declarations are not; all unqualified declarations in a collection definition belong to the element type, and declarations for the whole collection must be qualified with the collection type name.

```
Counter foo[15];

conc for(int i = 0; i < 15; i++)
  foo[i].count(i);

printf("%d
n", foo.sum());
```

Figure 25: Using a simple ICC++ collection

A `Counter[]` object can be used just like an array of `Counter`s, with the addition of its collection state. It is also declared just like one, with its size being specified with standard array syntax. Elements of a collection are accessed with `operator[]` just as array elements are. Collection members are used just like those of any other object. This is illustrated in Figure 25. Note that since there are no syntactic dependencies amongst the iterations of the `conc for` loop, the `count` calls can proceed in parallel.

4.1 Collection Members

Members can be defined for both the collection type and the element type just as they can be declared for normal classes. In addition, collections and collection elements have several pre-defined members that give information about their containing collection.

4.1.1 Pre-Defined Members

Both the collection class and element class have knowledge of the collection, and they provide several built in member functions that give information about the collection as a whole. The collection contains basic information like its size. Collection elements, unlike array elements, are inherently part of a collection, and so they provide functions that yield information about their containing collection. A list of these functions is given below.

- Collection members

  operator[ ](int) indexes the elements of the collection.

  operator[ ](void) returns an arbitrary collection element.

  size(void) returns the number of elements in the collection.

  nearest(void) returns the nearest element of the collection.

- Element members

  type_name::this is the enclosing collection of type `type_name`

  index(void) returns this element's index in the collection.

4.1.2 User-Defined Members

The member functions have access to the object state in the normal fashion: element methods access the state of the element and collection methods access the collection state. Note that, since the elements are distinct objects, their member functions do not have direct access to the collection state, which is in a different object. Nor do collection member functions have direct access to the state of the elements. However, the built in members provide collections and elements with information about each, allowing collections to exhibit an aggregate behavior through collaboration amongst the collection itself and the elements.

- Note that this integration of elements and collections is made possible because the two are defined together. If a collection's element type were separately defined, as it is for arrays, it could be used outside a collection and so it not have collection information built into it.

In Figure 24, `Counter[]` has two methods `count` and `sum`, which together implement a distributed counter, using the pre-defined members to pass information from the elements to the collection. In Figure 26, the elements gather data with `count`, and `elt_sum` recurses across the elements accumulating the total sum.

```
int Counter[]::sum(void) {
  return (*this)[0].elt_sum(size()-1);
}

int Counter::elt_sum(int last) {
  int my_index = index();
  if (my_index < last) {
    return elt_total + (*Counter[]::this)[my_index+1].elt_sum();
  else
    return elt_total;
}

int Counter::count(int val) {
  elt_total += val;
}
```

Figure 26: Defining collection member functions

4.1.3 Constructors

Both the element and collection types can define constructors. The syntax is just the same as constructors for normal classes. Constructors for the collection class can specify which constructor to call for the elements with the normal initializer notation for class fields. The collection in Figure 24 declared constructors for both the collection and the element class; in Figure 27, these constructors combine to set all the `elt_totals` and `total` to 0.

```
Counter::Counter(void) {
  elt_total = 0;
}

Counter[]::Counter[] : Counter() {
  total = 0;
}
```

Figure 27: Collection constructors

- C++ type conversion uses constructors with one argument as conversion operators. Collection constructors can be so used, but only for the outermost collection type. Since element types cannot exist without a collection, it makes no sense to construct them individually. Thus, only collection constructors will be used for conversions.

4.2 Nested Collections

Collections can be nested just as arrays can; nested collections are declared just as regular collections, with as many [ ] after the class name as desired. Internal layers of the collection are both collection objects and elements of another collection, hence they have both sets of pre-defined members. This is analogous to C++ multi-dimensional arrays, where internal arrays are both arrays themselves and elements of the enclosing array. Figure 28 shows how nested collections can be used to implement a matrix; nested collections are used to provide two-dimensional member addressing.

```
class Matrix[][] {
  int value;

 public:
  Matrix(void) { value = 0; }
  Matrix[](void) : Matrix() {};
  Matrix[][](void) : Matrix[]() {};

  int my_col(void) { return index(); }
  int my_row(void) { return Matrix[]::this->index(); }

  void Matrix[][]::invert(void) {
    int rows = size();
    int cols = (*this)[0].size();
    int diag = (rows + cols) / 2;
    conc for(int i = 0; i < rows; i++) {
      conc for(int j = i; i + j < diag; j++) {
        int temp = (*this)[i][j].value;
        (*this)[i][j].value =  (*this)[j][i].value
        (*this)[j][i].value = temp;
      }
    }
  }
};
```

Figure 28: A matrix collection

Nested collections can also be used to capture nested relationships; Figure 29 illustrates a simple grid data structure. Particles are contained in grid cells which make up a grid. The grid is one-dimensional to simplify the example. Since the nested structure is integrated, communication can move up and down it. The basic notion is that `move_particle` adjusts the positions of each particle, and then `regrid` moves any particles that have changed grid cells. The `move_particle` member communicate down the grid, which would be possible with normal arrays; however, the `regrid` member communicates up the structure, requiring the particles to know they are part of a nested collection.

4.3 Derivation

Collections can be derived from other collections and from standard classes. When deriving from other collections of equal nesting level, inheritance of members works just as it does for classes both for the collection and for the elements. When deriving from a collection of lesser nesting level, the innermost levels of the derived class inherit level-wise from the base class. When deriving a collection from a standard class, the element type is derived from that class.

C++ inheritance rules work *class-wise* in collection inheritance. For instance, C++ specifies that declaring a function `foo` in a derived class hides all functions of that name in the base class, not just any with the same type signature. For collections, declaring `foo` would hide all functions named

```
class ParticleGrid[][] {
 public:
  typedef ParticleGrid Particle;
  typedef ParticleGrid[] GridCell;
  typedef ParticleGrid[][] Grid;

 private:
  int mass;
  int pos;
  int velocity;
  int regrid_p;
  int GridCell::width;
  int GridCell::last_particle;
  int Grid::time_step;

 public:
  void move(void) {
   int inc = velocity * Grid::this.time_step;
   int width = GridCell::this.width;
   regrid_p = (pos / width - (pos += inc) / width);
  }

  void regrid(void) {
   Grid::this[GridCell::this.index()+regrid_p].add_particle(this);
   GridCell::this.remove_particle(this);
  }

  void GridCell::move_particles(void) {
   conc for(int i = 0; i < last_particle; i++)
     this[i].move();
  }

  void GridCell::add_particle(Particle p) {
   this[last_particle++] = p;
  }

  void GridCell::remove_particle(Particle p) {
   for(int i = p.index(); i < last_particle--; i++)
    this[i] = this[i+1];
  }

  void Grid::move_particles(void) {
   conc for(int i = 0; i < size(); i++)
     this[i].move_particles();
  }

  void Grid::regrid_particles(void) {
   conc for(int i = 0; i < size(); i++)
    conc for(int j = 0; j < this[0].size(); j++)
     if (this[i][j]->regrid_p) this[i][j].regrid();
    }
   }
  }
};
```

Figure 29: A particle-in-cell nested collection

foo for the element type, but not for the collection type. The reverse happens when a collection member function named foo is declared in the derived class. An example below elucidates this.

4.3.1 Derivation From Class

Figure 30 illustrates deriving a collection from a scalar class. The element type Paraccum has Accumulator as a base type, so the Paraccum constructor can call Accumulator's. The Accumulator simply counts all of the sum calls. The collection functionality provides an interface using sum(int) and computes the total sum with sum(void). That functionality must defined separately, as it is not inherited.

```
class Accumulator {
  int total;
 public:
  Accumulator(void) { total = 0; }
  int sum(int i = 0) { return total += i; }
}

class Paraccum[] : public Accumulator {
 public:
  Paraccum(void) : Accumulator() { };
  Paraccum[](void) : Paraccum() { };

  Paraccum[]::sum(int i) { return operator[]().sum(i); }
  Paraccum[]::sum(void) {
    int total = 0;
    conc for(int i = 0; i < size(); i++)
      total += operator[](i).sum();
    return total;
  }
};
```

Figure 30: Collection derived from class

This derivation also illustrates the *class-wise* nature of collection derivation alluded to above. The Paraccum[] class defines member functions called sum, and in C++ these would hide any functions called sum that would otherwise be inherited. However, since collection inheritance is class-wise, these functions only hide any functions called sum that Paraccum[] would inherit, but not Accumulator::sum which is still inherited by Paracum.

4.3.2 Derivation From Collection

In Figure 31, the collection Paraduce inherits from Paraccum. It extends the Paraccum to allow accumulation operations other than addition. Both the collection type and the element type are derived in the normal manner, so Paraduce inherits the total field from Paraccum and Paraduce[] inherits the sum(int) member from Paraccum[]. Also, Paraduce[] is a derived type of Paraccum[] and Paraduce is a derived type of Paraccum.

```
class Paraduce[] : public Paraccum {
 typedef int (*reducer)(int, int);
 reducer op;
 reducer Paraduce::op;

 public:
  sum(void) { return total; }
  sum(int i) { return total = op(total, i); }

  Paraduce(reducer iop) : Paraccum() { op = iop; }
  Paraduce[](reducer iop) : Paraduce(iop) { op = iop };

  Paraduce[]::sum(void) {
    int total = 0;
    conc for(int i = 0; i < size(); i++)
      total = op(total, operator[](i).sum());
    return total;
  }
};
```

Figure 31: Collection derived from collection

4.4 Distribution

While the ICC++ syntax makes little distinction between local and remote objects, locality must be managed very carefully to obtain reasonable performance. ICC++ provides a mechanism for distributing collections to permit both concurrency generation and locality management; it works by providing two views of a collection: a logical view and a physical view. The logical view allows members of the collection to be distributed in arbitrary ways, while the physical view provides the ability to write processor-centric loops and other low-level constructs.

4.4.1 The Physical View

All collections are distributed cyclically across the physical nodes of the underlying machine, starting from node 0. To allow processor-centric operations to be expressed conveniently, direct access to this physical distribution is provided by the following three operators.

physical_size(void) returns the number of elements in the physical collection layout, which is not guaranteed to be the same as the size returned by `size(void)`.

element(int) returns a physical element of the collection. These elements are layed out in the underlying cyclic distribution, so `element(3)` will be on node 3, for instance.

valid(void) returns `true` if a given element returned by `element(int)` is part of the logical collection, which is not guaranteed to be the case.

In order to use these primitives to implement processor-centric operations, one more piece of information is required: the number of processors in the system, which is provided in ICC++ by the variable `NUM_PROC`. A processor-centric member map operation is shown in Figure 32; this function loops through each collection element on a given node, calling a supplied member function.

```
void Collection[]::do_processor(int proc, void (*Collection::fun)(void)) {
 Collection irep;
 int idx = proc;
 int num_local_elts = physical_size() / NUM_PROC;
 while ((irep = this[idx+=NUM_PROC]).valid() && num_local_elts-- > 0)
  irep.*fun();
}
```

Figure 32: Processor-centric collection operation

This physical distribution corresponds to the default behavior of collections, but more complex distributions can be layered on top of it, as described in the next section.

4.4.2 The Logical View

Many applications require collection distributions other than cyclic to achieve acceptable performance. ICC++ has two ways of creating a logical view of a collection that provides a customized distribution: overloading `operator []` and *mapped collections.*

Overloaded `operator []` The default `operator []` defined for collection types directly uses the underlying cyclic distribution, but it can be overloaded to support customized layouts. Using `element`, overloaded versions of `operator []` can map the physical layout of the collection into any logical layout it fancies. For instance, to implement a blocked distribution, `operator []` can be defined as shown in Figure 33.

```
BlockCollection BlockCollection[]::operator[](int i) {
  int block_size = size() / NUM_PROC;
  int which_block = i / block_size;
  int block_offset = i % block_size;
  return element(which_block + block_offset*NUM_PROC);
}
```

Figure 33: Overloaded `operator []`

Note that to define distributions for nested collections, `operator []` must be overloaded at each level, as there is no `operator [][]` defined and no way to define it.

Mapped Collections Irregular distributions that do not fit cleanly into an overloaded `operator []` are support through the use of *mapped collections.* A collection map enumerates the distribution of collection elements across virtual processors. All maps are read from a file in an implementation dependent manner[5]; the file is a sequence of integers with the significance shown in Figure 34.

A collection created using a map is automatically provided with an overloaded `operator []` that accesses the collection in accordance with the map specified. The map is cycled through multiple

[5]For instance, an ICC++ compiler could have a global variable for the file name, it could be hardwired or specified as an option to the executable

```
Number of maps in file

Number of virtual processors for map 1
virtual processor number for element 0
virtual processor number for element 1
virtual processor number for element 2
...

Number of virtual processors for map 2
virtual processor number for element 0
virtual processor number for element 1
virtual processor number for element 2
...

Number of virtual processors for map N
...
```

Figure 34: Map file format

times for collections with more elements than the map specifies. The member `element(int)` still has access to the underlying cyclic distribution.

```
MAP_FILE
--------

1

4
2
0
0
1
```

PE 0	PE 1	PE 2	PE 3
1	3	0	
2	7	4	
5	11	8	
6	15	12	
9			
10			
13			
14			

Figure 35: A map file and the resultant distribution of a 16-element collection.

Mapped collections are created by supplying the map index as a placement argument to the creation of the collection; the example in Figure 35 illustrates the format of the map file. It has one map, which has 4 virtual processors. Collection element zero is mapped to node 2, elements 1 and 2 to node 0 and element 3 to node 1. This is repeated for collections with more than four elements. A collection using this map would be created as `new (1) CollectionClassName[16]`, as 1 is the (one-based) index of the map in the file.

4.5 Summary

Collections generalize arrays and integrate them into the object model. This integration allows methods to be defined upon arrays, and array elements to cooperate in implementing an aggregate behavior. Furthermore, since collections are collections of objects, they are part of the concurrency control model, and thus can be used to express parallelism.

A Performance Annotations

Efficient execution on parallel machines generally requires careful management of object placement and the ability to optimize concurrency control. While some of this can be done by compiler analysis, it can be simpler for the programmer to provide such information, as (s)he presumably has a high-level notion of how the program works. ICC++ provides the type specifiers `restrict` that make assertions about object aliasing. There is also a mechanism to provide hints regarding object locality.

A.1 `restrict`

The `restrict` type specifier can be applied only to object declarations, and it asserts that, while the given declaration is in scope, no reference will be made to its referent anywhere in the program except through the given name. The `restrict` specifier applies only to the referent itself; it is not transitive to the objects pointed to by the referent.

```
class Tree {
  Node data;
  Tree *left;
  Tree *right;

 public:
  void traverse(void (*Node::fun)(void)) {
   restrict Tree *l = left;
   restrict Tree *r = right;
   conc {
    data.*fun();
    left- >traverse();
    right- >traverse();
   }
  }
};
```

Figure 36: `restrict`ing subcomputations

Figure 36 uses `restrict` to assert that the two calls to `traverse` will never refer to the same portion of the tree, and that nobody else will touch the tree while `traverse` is executing.

A.2 Object Locality

The locality annotations of ICC++ have two components: allocation annotations and usage annotations. The allocation annotations tell the system to create an object locally with respect to the creator. The usage hints tell the system when an object will be local to the calling method. Both of these constructs take a relative approach to locality, and so can only be used within member functions, where the creator has a definite location[6]. These two portions work together as a contract; the programmer must ensure that, if the system obeys all the allocation annotations, the usage annotations will be correct.

[6]A function has no associated object and so the notion of local to a function makes no sense.

A.2.1 Allocation

ICC++ provides a pre-defined placement parameter for `operator new` named `LOCAL` which directs the system to create the object in the same place as the calling method. It is used in the normal manner for placement syntax, as shown below.

```
foo *a = new (LOCAL) ObjectClass(7);
```

In addition to this way of asserting locality, all objects created as auto variables of member functions have the locality hint by default.

A.2.2 Usage

Usage locality is asserted with the type specifier `local`, applied to the object declaration. This is a hint in that the system may ignore it, but it must be correct if the system obeys the creation locality directives.

```
int foo::bar(local foo *a)
  local foo *b = get_my_buddy();
  work(a, b);
```

Here the `work` member function can be specialized to assume that both its parameters are local to `this`.

References

[1] A. A. Chien, U. S. Reddy, J. Plevyak, and J. Dolby. ICC++ – a C++ dialect for high performance parallel computing. Submitted for Publication, 1995.

[2] Andrew Chien and Uday Reddy. ICC++ language definition. Concurrent Systems Architecture Group Memo, February 1995.

[3] John R. Ellis and David L. Detlefs. Safe, efficient garbage collection for c++. Technical report, Xerox Palo Alto Research Center, June 1993.

[4] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[5] 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.

[6] Bjarne Stroustrup. *The Design and Evolution of C++.* Addsion-Wesley, 1994.

[7] Sun Microsystems Computer Corporation. *The Java Language Specification*, March 1995. Available at http://java.sun.com/1.0alpha2/doc/java-whitepaper.ps.

Index