Doc. No.: 00-0025/N1248
Date: 19 April 2000
Author: Performance WG

Technical Report on Performance Issues

(*extremely rough draft 1999-09-27*)

(plus extremely rough revisions by Lois Goldthwaite 2000-04-15)

Contents:

Preamble

- Definition of terminology and scope of the report
  - Description of potential resource limitations
  - Kinds of problems often encountered in resource-limited environments
  - Offer criteria used in the selection of an appropriate programming language

"Performance" has many aspects -- execution speed, code size, data size, memory footprint at run time, or time and space consumed by the edit/compile/link process. It could even refer to the time necessary to find and fix code defects. Most people are primarily concerned with execution speed, although code footprint and memory usage can be critical for small embedded systems where code is burned into ROM, or where ROM and RAM are combined on a single chip.

An old rule of thumb is that there is a trade-off between program size and execution speed -- that techniques such as declaring code **inline** make the program larger but faster. But now that processors make extensive use of on-board cache and instruction pipelines, the smallest code is often the fastest as well.

Compilers typically use a heuristic process in optimizing code, and it may be different for small and large programs. Therefore it is difficult to recommend any techniques which are guaranteed to improve performance in all environments. It is vitally important to measure a performance-critical application in the target environment and concentrate on improving performance where bottlenecks are discovered.

A Guide for Users

- (*Note*: Must be clear, crisp, and comprehensible.)
- (*Note:* Start with a summary of the C++ object model)
- Assuming all features are available, routes for performance improvement
- Information from the EC++ style guide as starting point
- Qualitative analysis of the advice given; offer examples with some associated numbers (time, memory, size, etc).
- How to write efficient templates.

A Guide for Implementors

Performance effects of language features and combinations thereof

Does the C++ language have inherent complexities and overheads which make it unsuitable for performance-critical applications? For a program written in the C-conforming subset of C++, will penalties in code size or execution speed result from using a C++ compiler instead of a C compiler? Does C++ code necessarily result in "unexpected" functions being called at runtime, or are certain language features, like multiple inheritance or templates, just too expensive (in size or speed) to risk using? Do these features impose overheads even if they aren't explicitly used?

Efficiency has been a major design goal for C++ from its earliest days, also the principle of "zero overhead" for any feature which is not used in a program. If there are places where C++ cannot guarantee zero overhead for unused features, this paper will attempt to document them.

- Overheads from Inheritance
- Overheads from Exception Handling
  - Essential elements of all Exception Handling methods
  - Myths and Reality of Exception Handling Overhead
  - Predictability of Exception Handling Overhead
- Overheads of Template Code
- Overheads of Namespaces:

  Namespaces do not add space or time overheads to code. They do, however, add some complexity to the rules for name lookup, and they add more characters to a program's source code (if only "using namespace std;"). Their advantage is that they provide a mechanism to partition large projects and so avoid name clashes.

- Overheads of New-style Casts:

  Of the four new-style casts added to Standard C++, three of them have no size or speed penalty. dynamic_cast<> does incur a small amount of overhead at run time, although the mechanisms necessary to implement it are shared with RTTI. The syntax of new casts is easier to identify in source code, and may thus contribute to more correct programs.

- The **restrict** keyword:

  The **restrict** keyword in C99, which allows the compiler to optimize code without worrying about aliasing of pointers, has been shown to make

dramatic improvements in execution speed. Some experiments show run times reduced by half, compared to equivalent code compiled without support for **restrict**. Compiler vendors targeting high-performance applications might want to implement **restrict** as an extension.

- Standard template ABI?
- Compile-time evaluation, like locales in libraries.
- "Piggy-backing" off offerings of other languages/systems to accentuate those of C++.
- Garbage collection: code generation, library design.
- "zero-overhead principle": citing of those language features that deviate from this principle.

Performance effects of library features and combinations thereof

Overheads from Locales

Basic I/O hardware addressing (latest draft notes ...)

ROM-ability

Definition of volatile

Performance guarantees

Programmer-directed Optimizations

-- Description of current approaches
-- Code generation control, including memory placement, initialization characteristics, et al.
-- #pragma, restrict, other language modifications
-- Application of measurement results in making choices.
-- Transforming virtual calls into non-virtual calls
-- Alternatives to exception handling
-- Effects of restrictions upon character types
-- Characterization of performance guarantees

-- Coding style can Affect Performance

Library Extensions

-- Basic I/O hardware addressing (WG21/N1192)
-- Fixed-size data types (as in C9X's)

Overheads from Inheritance

Overhead examples

- Run-time type identification (RTTI),
- multiple inheritance,
- virtual template member functions,
- virtual inheritance
- class hierarchies,
- Unnecessary costs for empty base,

RTTI overheads

- Typically, a pointer to a type_info object is stored in a class's vtbl. RTTI can only be used with classes which have at least one virtual function.
- One typical implementation costs one table per class; enough storage for class name ("**typeid**") plus five words
- In other words, something like 40 bytes times the number of classes in the application.
- Often, RTTI is used with **dynamic_cast;** this requires exception-handling (EH). (Some implementations do allow RTTI without EH
- Whole-Program Analysis (WPA) can help; there is no need to generate RTTI tables for types not tested
- How about "partial-program analysis", such as "**final**"? "This is my program, it's not a library for others"?
- Relevance of "Vortex" system? (see Mike Ball)
- Don't forget, the savings for small embedded applications is multiplied by number of devices targeted for the production run.
- **inline** is a PDO; so is usage of non-virtual functions.

Overheads of Inheritance

- **A class without any virtual functions is equivalent to a simple C struct.** The size of an object of the class is the sum of the sizes of its data members, (plus any padding necessary
- **Does inheritance by itself add overheads?** In a typical implementation, data members of a base class will occupy space at the beginning of an object of a derived class. This need not cost any more data space than the alternate design of creating a data member of the base class type. In the simplest case inheritance may save in code size and execution speed, since delegating functionality to a member object requires pass-through functions in the containing class. Calls to non-virtual functions are

resolved at compile time, so there is no run time penalty from single inheritance.

- **Do virtual functions add overheads?** Calls to virtual member functions are resolved at run time, depending on the dynamic type of the object. In a typical implementation, each object in the hierarchy acquires an extra data member, a vptr, pointing to a vtbl listing the appropriate version of virtual functions for that class type. So the cost of virtual functions is an extra data pointer per object, plus a vtbl per class.

  At run time, there is a cost of calling the virtual function by indirection through the vptr, indexing into the vtbl, and calling a function through a pointer. This cost, in a typical implementation, adds approximately xxx *[at a guess, 3 - 10]* instruction cycles per call, compared with direct calls to a class-specific function, resolved at compile time. Alternate mechanisms of determining the appropriate function to call, such as an **if** statement or **switch/case** block, also have their overheads, however. If a virtual function is called repeatedly inside a tight loop, a possible programmer-directed optimization is to determine the runtime type of the object outside the time-critical section, and use class-specific direct calls inside the loop.

  [Q for compiler writers -- Does the vtbl add to code size, or static data size? Is there a vtbl per translation unit? Can we give advice to implementors here? Or to programmers? Does a base class with all virtual functions defined inline result in vtbl bloat? If the programmer defines at least one virtual function out-of-line, does that solve the problem? Are there special considerations when virtual function tables appear in shared libraries?]
- **The principal disadvantage of virtual functions is that they prevent the compiler from inlining code, since the type of the object won't be known until run time.**
- ***Here are my notes on Bjarne's tests:***
  - BS – experiments to test overhead of virtual functions, vfs on left branch of tree, vfs on right branch
  - Static function calls are slightly faster than ordinary member funcs (less than 25%)
  - No sig diff in runtime speed between ordinary function calls, virtual func calls, and virtual function calls among different branches of MI.
  - Func calls are cheaper than they used to be (compared to inline)
  - Virtual func calls are cheaper relative to ordinary function calls than they used to me
  - Tested on three compilers (MS, Borland, EDG (?))
  - Downcasts cost between three and four function calls. Independent of single or multiple inheritance, of which branch of MI, or of depth of MI. (looking at the algorithms, it’s probably executing those function calls).

- Crosscasts are more expensive. A crosscast costs between 6 and 50 times a single function call, depending on the compiler. They vary with how deep you start and finish in the hierarchy. Each level adds about 60% to overhead.
- These figures are a lot better than they were a couple of years ago.
- People should make less use of inlining these days.
- Later – forcing code out of cache, virtual func call (through a pointer) had overhead of 20% compared to plain function call. Maybe even 30% if you do a lot of other work in the loop and in the function call and then factor it out. But still no overhead in MI itself.

Multiple Inheritance Overheads

- Properly implemented, multiple inheritance should have very little extra cost. *[Bjarne ran some rough experiments at the Hawaii meeting. IIRC, his stats showed that multiple inheritance adds about 3 or 4 cycles to each function call, just the cost of indirection. He found no significant difference in calling funcs inherited from the 'left' side of the tree vs the 'right' side of the tree.]*
- There is "offset adjustment" in virtual calls
- Using "thunks" for virtual calls should eliminate any overhead for classes that aren't multiply inherited. (?)
- A pointer to a virtual member function requires an extra adjustment, but it is really minor.
- Virtual base classes add

[Mike can find literature references]

Virtual Template Function Overheads

- Virtual functions of a template class can create an overhead.
- Consider a template named **facet** which has a virtual member function **numput.**
- Every time **facet** is instantiated it generates virtual member functions.
- A bad library implementation could produce hundreds of kbytes.
- It's a library modularity issue: putting code into the template when it doesn't depend on template parameters, when it could be separate code, may cause each instantiation to contain large redundant code sequences. Suggestion: use non-template helper functions. (Another PDO.)

Virtual Inheritance Overheads

[??? No notes]

Class Hierarchies Overheads

[??? No notes]

Unnecessary costs for empty base Overheads

[??? No notes]

Overheads from Exception Handling

Essential elements of all exception handling implementations.

- `try`

  Establish context for associated catch clauses.

- `catch`

  Run-time type information for finding catch clauses at throw time.

  > Overlapping but not identical information to that needed by RTTI features for thrown types. Must be able to match derived classes to base classes even for types without virtual functions, and to identify built-in types such as int. Conversion from base to derived classes (down-casting) not needed.

- Cleanup of handled exceptions.

  Exceptions which are not re-thrown must be destroyed upon exit of the catch block.

  "Magic memory" for exception object must be returned to the exception-handling system.

- Automatic and temporary objects with non-trivial destructors.

  Destructor must be called if an exception occurs after construction and before destruction, even if no try/catch is present.

- All objects with non-trivial constructor and destructor.

  All completely-constructed base classes and sub-objects must be destroyed if an exception occurs.

- `throw`

  "Magic memory" must be allocated to hold a copy of the exception object

  exception object must be copied.

Closest matching catch clause must be found.

Intervening destructors must be executed.

- Enforcing exception specifications.

  Conformance of thrown type to list of specified types must be checked.

  Unexpected handler must be called if a mismatch is detected.

  A similar mechanism to the one implementing try/catch can be used.

- Operator new.

  Corresponding operator delete must be called if an exception is thrown from constructor.

  A similar mechanism to the one implementing try/catch can be used.

Implementations vary in how costs are allocated across these elements. Two typical strategies are the "dynamic" and "static" approaches.

The "dynamic" approach.

- `try`

  Save execution environment and reference to catch code on EH stack at try block entry (by calling setjmp or equivalent).

- Automatic and temporary objects with non-trivial destructors.

  Push constructed objects with address of their destructors onto a stack for later destruction. Remove them upon destruction. Typical implementations use a linked list for the stack.

- All objects with non-trivial constructor and destructor.

  One known implementation increments a counter for each base class and sub-object which is constructed. If an exception is thrown during construction the counter indicates which parts need to be destroyed.

- `throw`

  Pop objects from the stack and destroy them until a reference to catch code is found.

Restore execution environment of nearest handler (by calling longjmp or equivalent).

Advantages: simple, portable, and compatible with C backends.

Disadvantages: stack space and run time costs for try block entry and bookkeeping for auto and temporary objects as the EH stack is modified.

[One vendor reports speed impact of about 6% for a C++ to ANSI C translator. Another vendor reports that speed and stack space impacts can be greatly reduced by fine-tuning the code for saving the execution environment and doing object bookkeeping - N.B. these are *strictly* off-the-cuff estimates]

The "static" approach.

Translator generates read-only tables for locating catch clauses and objects needing destruction.

- `try`

  No runtime cost. All bookkeeping pre-computed as a mapping between program counter and code to be executed in case of an exception. Tables increase image size but may be moved away from working set to improve locality. Tables can be placed in ROM, and remain swapped out on VM systems until an exception is actually thrown.

- Automatic and temporary objects with non-trivial destructors.

  No runtime cost, same reasons.

- All objects with non-trivial constructor and destructor.

  No runtime cost, same reasons.

- `throw`

  Search tables to locate matching handlers and intervening objects needing destruction.

Advantages: no stack space or run time costs for try/catch and object bookkeeping.

Disadvantages: more difficult to implement.

[One vendor reports a code and data space impact of about 15% for the generated tables. This is an upper limit, since in the vendor’s environment there was no need to reduce the

image size of programs as long as the working set wasn't increased. N.B. these are *strictly* off-the-cuff estimates.]

Exception specifications.

The need to enforce specifications at runtime has costs as described above. However, they can allow optimization of other code by making catch clauses unreachable and violations of other exception specifications impossible. Empty throw specifications are especially helpful for optimization.

The "you don't pay for what you don't use" principle.

Exception handling in general imposes costs even if it is not used. For example, a function which constructs automatic objects and then calls a function which cannot be proven by the compiler not to throw an exception will incur object bookeeping (in the static approach, data space, in the dynamic approach, runtime and stack space). An optimization is available, however, with the static approach: exception tables and runtime support code can be stripped at link time if no exceptions are thrown in an entire program. This would eliminate all costs associated with EH.

Other error-handling strategies.

All approaches to error-handling including error-return codes, global error values, process termination and ignoring errors have associated costs in run time, data space, program correctness maintenance and readability. In evaluating the costs of exception-handling the costs of the alternatives should not be ignored.

Missing stuff.

There were some items discussed in the working group which we were unable to flesh out. These included:

Advice to implementors, specifically references to literature on EH (e.g. 'C' Language Translation).

Potential implementation pitfalls.

A comparison of the costs of other strategies.

Myths and Reality of Exception Handling Overhead

Preliminary remarks

Exception Handling provides a systematic and robust approach to error handling. As opposed to the traditional C style of indicating run time problems by returning an error code, which must be checked at every point a function is invoked, EH isolates the rare problem-handling code from the normal flow of program execution. Automatic destruction of stack objects when an exception is thrown renders a program less likely to leak memory or other resources. And with EH, once a problem is identified, it can't be ignored -- failure to catch and handle an exception results in program termination.

Early implementations of Exception Handling resulted in sizable increases in code size. This led some programmers to avoid it and compiler vendors to provide switches to suppress the feature. In some embedded and resource-constrained environments, EH was deliberately excluded.

It is difficult to discuss about EH overheads without a rough idea about possible implementations.

Distinguish between

- **Try overhead**: data and code that must be generated for and/or executed at try/catch time (that is getting ready for catching exceptions that may never occur): this is actually the true overhead.
- **Regular function overhead**: data and code that must be generated for and/or executed by the functions which do not invoke themselves any exception related feature (breaking the "pay as you go" principle)
- **Throw cost**: data and code that are generated and executed when throwing an exception. This can hardly be regarded as an overhead! But there may be different implementations, with different cost, the value of which depends on various criteria.

Compile-time overhead

- Compilation is more difficult, depending on the complexity of the implementation
- Some compile-time optimizations may become trickier (or even impossible?): we *need examples*

Run-time Overhead

Two main strategies: setjmp model and table model..

Space Overhead

- The size of the objects does not need any modification
- EH implies a (weak) form of RTTI, thus increasing the code size.
- Setjmp model implies code generation for try/catch
- Table model implies *static* data generation
- Setjmp model implies *dynamic* data structures to
- Handle the jmp_buf environments and their mapping to catches
- Register the local objects to be destroyed
- Handle the throw specifications of the functions which have been called

Time Overhead

Setjmp Model

- At try/catch time
- Stack the jum_buf environments
- When calling regular functions
- Register the functions that are called (for throw spec checks)
- Register the local objects when they are created
- During a throw
- Find the environment to do a longjmp to (this involves some RTTI-like check)
- Destroy the registered local objects
- Check the throw specifications of the functions called in-between

Table Model

- At try/catch time
- No overhead at all
- When calling regular functions
- No overhead at all
- During a throw
- Go upwards the stack frame, and for each frame
- Check in the table whether we have a catch clause for the exception at this frame level (this involves some RTTI-like check) and if so execute it
- Otherwise,
    - Locate the corresponding function in the static table (O(log F), F being the number of functions)
    - Destroy the constructed local objects (the static offsets of which are found in the table, the set of which depends on the program counter value)
    - Check the throw specification

Predictability of Exception Handling Overhead

Prediction of throw/catch performance

In the Embedded C++ rationale (http://www.caravan.net/ec2plus/rationale.html), one of the reservations expressed about EH is the unpredictable time that may elapse after a throw and before control passes to the catch clause, while automatic objects are being destroyed. It is important in some systems to be able to predict accurately how long operations will take. *[I don't know how to address this issue. I don't see how it's different from estimating the time taken to destroy automatic objects at the end of a scope. But then I don't know anything about embedded programming.]*

Another reservation in the EC++ rationale concerned the memory footprint of the necessary data structures.

Empty throw spec considerations

Can empty throw specs help a compiler produce more optimal code?

It should reduce overhead to zero, if called functions cannot throw *[and some current compilers are able to do this, when given an empty throw spec]*.

However, a poor implementation can produce worse code when it produces an extra try-catch for functions that don't need it.

Example:

```
int g() throw();

void f() {
    int n = g(); --->
    // Rewritten like this …
    // int n;
    // try {
    //     n = g();
    // } catch (...) {
    //     terminate();
    // }
}
```

Comparisons between "Table" model and "Longjmp" model

Some implementations support both the "Table" model and the "Longjmp" model, such as Edison Design Group (?) or Cygnus (?). Maybe some comparisons could be made from their generated code.

Table Model

- Discuss implementation complexity.
- Some of the job is front-end, parsing the language.
- Most of the job is back-end, building tables, intermediate representations, 1-2 man-months (?).
- Another cost of EH is its interaction with optimization levels. Often it increases the bug level of the higher optimization levels.
- Another factor is predictability. Especially for small to medium embedded apps, it's important to be able to estimate resources. What assistance can we give with ability to make accurate estimates for time and space?

How do we characterize application areas?

Embedded Systems:

We consider there are the following types of consumer based application areas.

- small
  use single chips(include ROM/RAM in a chip)
  Example: Engine Control for Automobile
- medium
  use external ROM/RAM, the size is limited
- large
  use external ROM/RAM, the size is unlimited

SCALE	RAM	ROM	TIMING
small (engine control)	32K	256K (program code 128K)	minimum cycle of engines(3msec) (ex. 10000rpm, 4cylinders)
medium (digital handy VCR)	1M (program data 32K)	256K-512K	1 refresh time(8msec for the vertical)
large(PDA)	2M(minimum)	2M(minimum)	N.A.(depend on user's sense)

Servers:

We consider there are the following types of server application areas.

- Small
  32 MB RAM (?)
- medium
  256 MB RAM (?)
- large
  (?)

In server applications, the performance-critical resources are typically speed (transactions per second?), and working-set size (which also impacts throughput speed).

Overheads from Using Templates

Template overheads

- **Do templates cause code bloat?** Template classes or functions will generate a new instantiation of code every time a different argument type (or combination of types) is used in the program. This can potentially lead to an unexpectedly large code size. A typical way to cause this problem is to create a number of Standard Library containers to hold pointers of various types. Each type causes an extra set of code to be instantiated.

  In an experiment *[run by Tom Plum in Hawaii]* a program instantiating 100 instances of std::list of pointers to a single type was compared with a second program instantiating list<T*> for 100 different types of T. These programs were compiled with a number of different compilers and command-line options. With one compiler, the second program produced code over 19 times as large as the first program. With a different compiler, the first program was larger by a factor of less than 3. *[I still have all the numbers in my notes, if anyone wants them, but it would probably be better to run a new series of carefully-controlled experiments.]*

  It is possible for the compiler or linker to perform this optimization automatically *[albeit with longer build times]*, but without tool support, optimization can also be performed by the Standard Library implementation or by the application programmer. If the compiler supports partial specialization and member template functions *[hope I've got that accurate]*, the library implementor can partially specialize containers of pointers to use a single underlying instantiation for void *. This technique is described in C++PL 3rd ed *[and has been implemented by H--- H---- in the M---- library, with excellent results]*. As a programmer-directed optimization, it is possible to write a template class called, perhaps, plist<T>, containing a list<void *> member to which all operations are delegated. Source code must then refer to plists rather than Standard lists, so the technique is not transparent, but it is workable.

Templates vs Inheritance

- Any non-trivial program needs to deal with data structures and algorithms. Because data structures and algorithms are so fundamental, it's important that their use be as simple and error-free as possible.
- The template containers in the Standard C++ Library are based on principles of generic programming, rather than the "object oriented" approach used in other languages such as Smalltalk. An early set of

foundation classes for C++, called the National Institutes of Health Class Library, was based on a class hierarchy in the Smalltalk tradition. *[See Data Abstraction and Object Oriented Programming in C++, by Keith Gorlen, et al., 1990. As there were no "standard" C++ classes in the early days, and because NIHCL was freely usable, having been funded by the US Government, it had a lot of influence on design styles in C++ in subsequent years.]* Of course, this was before compilers could handle complicated uses of templates.

- In the NIH library, all classes in the tree inherit from a root Object class, which defines interfaces for identifying the real class of an object, comparing objects, and printing objects. *[The Object class itself inherits from class NIHCL, which encapsulates some static data members used by all classes.]* Most of these functions are virtual, and must be overridden by derived classes. The hierarchy also includes a Class class, to provide a library implementation of RTTI (which was not then part of the language). The Collection classes, themselves derived from Object, can hold only other objects derived from Object which implement the necessary virtual functions. Here is a portion of the hierarchy tree from NIHCL (taken from the README file):

```
NIHCL - Library Static Member Variables and Functions
    Object - Root of the NIH Class Library Inheritance Tree
        Bitset - Set of Small Integers (like Pascal's type SET)
        Class - Class Descriptor
        Collection - Abstract Class for Collections
            Arraychar - Byte Array
            ArrayOb - Array of Object Pointers
            Bag - Unordered Collection of Objects
            SeqCltn - Abstract Class for Ordered, Indexed
                      Collections
                Heap - Min-Max Heap of Object Pointers
                LinkedList - Singly-Linked List
                OrderedCltn - Ordered Collection of Object
                              Pointers
                    SortedCltn - Sorted Collection
                        KeySortCltn - Keyed Sorted Collection
                Stack - Stack of Object Pointers
            Set - Unordered Collection of Non-Duplicate Objects
                Dictionary - Set of Associations
                    IdentDict - Dictionary Keyed by Object
                                Address
                IdentSet - Set Keyed by Object Address
        Float - Floating Point Number
        Fraction - Rational Arithmetic
        Integer - Integer Number Object
        Iterator - Collection Iterator
        Link - Abstract Class for LinkedList Links
            LinkOb - Link Containing Object Pointer
        LookupKey - Abstract Class for Dictionary Associations
            Assoc - Association of Object Pointers
            AssocInt - Association of Object Pointer with Integer
        Nil - The Nil Object
        Vector - Abstract Class for Vectors
            BitVec - Bit Vector
            ByteVec - Byte Vector
            ShortVec - Short Integer Vector
            IntVec - Integer Vector
            LongVec - Long Integer Vector
```

```
                FloatVec - Floating Point Vector
                DoubleVec - Double-Precision Floating Point Vector
```

- Thus the KeySortCltn class, roughly equivalent to std::map, is seven layers deep in the hierarchy:

```
NIHCL
    Object
       Collection
           SeqCltn
               OrderedCltn
                    SortedCltn
                        KeySortCltn
```

- Because a linker cannot know which virtual functions will be called at runtime, it typically includes the functions from all levels of the hierarchy in the executable program. This can lead to code bloat without templates.
- There are other performance disadvantages to "object oriented" collection classes. One of these is that primitive types cannot be inserted into the collections. These must be replaced with classes in the Object hierarchy which are programmed to have similar behavior to primitive arithmetic types, such as Integer and Float. This circumvents processor optimizations for arithmetic operations on primitive types. It is also difficult to provide exact duplicates of arithmetic behavior through class member functions and operators.
- Because C++ has compile-time type checking, providing type-safe containers for different contained data types requires code to be duplicated, for the same reason that template containers are instantiated multiple times. To avoid this duplication of code, the NIHCL collections hold pointers to the base Object class, a generic type. However, this is not type safe, and requires run-time checks to ensure objects are type compatible with the contents of the collections. It also leads to many more dynamic memory allocations, which can hinder performance.
- Because classes to be used with the NIHCL must inherit from Object and implement a number of virtual functions, this solution is intrusive on the design of classes from the domain. The C++ Standard Library containers do not impose such requirements on their contents. [A class used in a container must be Assignable and CopyConstructible; often it additionally needs to have a default constructor and implement operator== and operator<.] The obligation to inherit from class Object often means that using Multiple Inheritance becomes necessary for this reason alone, since domain classes may have their own hierarchical organization.
- The C++ Standard Library lays out a set of principles for combining data structures and algorithms from different sources. Inheritance-based libraries from different vendors, where the algorithms are member functions of the containers, can be difficult to integrate and difficult to extend.

Basic I/O hardware addressing.

Outcomes:

Document 1:

Basic I/O hardware addressing - standardized syntax

- Introduction
- Conceptual model for I/O registers
- *Access_type* characteristics
- Header *iohw* overview
- Constraints
- Detailed function description

Uses the same editorial layout and formatting conventions as the C++ standard.

Document 2:

Basic I/O hardware addressing - Guide for implementors

- Abstraction model
- Typical access methods and considerations
- Examples of *access_type* descriptors (illustrate implementation principles)
- Examples of template implementations (illustrate implementation principles)
- Complete *iohw* header implementations for typical processor architectures.

Work plan:

Wants to implements the basic I/O hardware addressing functions described in document N1192 in practice. Most of the work will be to consider implementation strategies for the abstract *access_type* which is used both as a identification of a register object and as a complete description of how a given register should be addressed in the given platform.

Step 1. : Exploit the implementation limitations

Addressing test cases:

- **Address is defined at compile time.** The address is a constant. This is the most simple case and also the most common case with smaller architectures.

- **Address is initiated at runtime**. Variable base address + constant offset. I.e. the *access_type* must contain an address pair (address of base register + offset address).
- **Indexed bus addressing** (also called orthogonal or pseudo bus addressing). Operates in this way: First the pseudo address is written to the index port register located at a given address, then the data operation(s) is done at the following port register address. I.e. the *access_type* must contain an address pair (index address, register address)
- **Access via user defined access driver functions**. Are typical used with larger platforms and with small single chip processors (ex. to emulate an external bus). I.e. *access_type* must contain pointers or references to access functions (read, write, and, or)

Address interleave

An *access_type* parameter which becomes relevant when the data bus size of the I/O chip is smaller than the data bus of the system.

If for instance a processor architecture has a byte aligned addressing range and a 32 bit processor data bus, and a 8 bit I/O chip is connected to the 32 bit data bus, then three adjacent registers in the I/O chip will have the processor addresses:

```
<addr + 0>, <addr + 4>, <addr + 8>
```

This can also be written as:

```
<addr + interleave*0>, <addr+interleave*1>, <addr+interleave*2>
```

where *interleave* = 4.

Interleave will be a constant for the given I/O register definition.

Register endian

An *access_type* parameter which becomes relevant when the data bus size of the I/O chip is smaller than the size of the I/O register. Forces the access operation to be split over two or more register addressing operations. The endian parameter describes whether the MSB or the LSB byte of the I/O register is located at the lowest access address.

Register endian will be a (bool) constant for the given I/O register definition.

More than one addressing range

The same I/O function syntax are used with different address busses.

access_type initialization

Is relevant when the address is initiated at runtime (address case 2) and probably also for a set of access driver functions (address case 4) (Use *access_type* constructor ?)

Extended syntax checking.

To detect illegal use of access types. For instance write of a 16 bit variable to an 8 bit register, or read of a write-only register.

This part of the work plan can be implemented and tested using access stubs. Creates a template frame work for implementation with a given processor architecture.

Step 2: Implement <iohw> header for two or three different processor architectures.

- Select processor families.
  - The following CPU families have been suggested : 80x86 (two different addressing ranges, two kind of pointers), H8/300 (simple linear addressing ranges, special interleave, register bit addressing), V830 (extended interleave and endian requirements)
- Select compilers for test (gcc compilers are candidates, others are very welcome)
- Implement *iohw* header with full machine code generation. Check the code generation efficiency for I/O register access.
- Look at how to get debug information for accessing I/O registers directly with debuggers. i.e. for use during non-runtime.

Step 3: Make document drafts

Draft Basic I/O hardware addressing - Guide for implementors document

Draft Basic I/O hardware addressing - standardized syntax document

Programmer Directed Optimizations

Much of the following is inspired / borrowed / plagiarized from various sources, not all of whom are credited. Dan Saks, Dov Bulka, and Scott Meyers are certainly among the inspirations. If any of them thinks portions of this text are too close to their copyrighted material, I will be glad to rewrite the offending portions. A lot of it is from my notes scribbled at lectures.

Programmers are sometimes surprised when their programs call functions they haven't specified, maybe even haven't written. Understanding what a C++ program is doing is important for optimization.

- Shift expensive computations from the most time-critical parts of a program to the least time-critical parts (often, but not always, program start-up).
- Whenever possible, compute values and catch errors at translation time rather than run time.
- Know what functions the C++ compiler silently generates and calls. Simply defining a variable of some class type may invoke a potentially expensive constructor function. As a general principle, don't define a variable before you are ready to initialize it. This prevents initializing the variable twice.
- In constructors, prefer initialization of data members to assignment. These are the steps taken to construct a variable of class type: first, any assignments specified in the member initialization list are performed. Next, other members of class type (but not primitive types) are initialized by their default constructor, if one is available. Only then is the body of the constructor executed.
- Passing arguments to a function by value *[void f(T x)]* is cheap for built-in types, but potentially expensive for class types, since the copy constructor may be non-trivial. Passing by address *[void f(T * x)]* is light-weight, but changes the way the function is called, and exposes the passed object to modification by the called function. Passing by reference-to-const *[void f(T const & x)]* combines the safety of passing by value with the efficiency of passing by address. But be careful not to create unnecessary temporary objects, by using an argument which must be converted to the type of the function parameter.
- Unless you need automatic type conversions, make all one-argument constructors **explicit**. This will prevent calling them accidentally. Conversions can still be done by specifying them in the code, without performance penalty.
- Understand how and when the compiler generates temporary objects. Often small changes in coding style can prevent the creation of temporaries, with beneficial effects on run time speed and memory footprint. Temporary objects may be generated when passing parameters to functions, returning values from functions, or initializing objects. Sometimes it is helpful to widen a class's interface with functions that take different data types to prevent automatic conversions (such as adding an overload on char * to a function which takes a std::string parameter).

- Rewriting expressions can reduce or eliminate any need for temporary objects. If **a**, **b**, and **c** are objects of class **T**:

```
T a;              // inefficient: don't create an object
                  // before its initialization is ready
a = b + c;        // inefficient: (b + c) creates a
                  // temporary object and then assigns it
                  // to a
T a( b ); a += c; // no temporary objects created
```

- Use the return value optimization to give the compiler a hint that temporary objects can be eliminated. The trick is to return constructor arguments instead of objects, like this:

```
const Rational operator * ( Rational const & lhs,
                            Rational const & rhs )
{
    return Rational( lhs.numerator() * rhs.numerator(),
                     lhs.denominator() * rhs.denominator() );
}
```

  Less carefully written code might create a local Rational variable to hold the result of the calculation, then use the assignment operator to copy it to a temporary variable holding the return value, then copy that into a variable in the calling function. But with these hints, the compiler is able to construct the return value directly into the variable which is specified to receive it.

- Prefer pre-increment and -decrement to postfix operators. Postfix operators (i++) copy the existing value to a temporary object, increment the internal value, then return the temporary. Prefix operators (++i) increment the value and return a reference to it. With objects such as iterators, creating temporary copies is expensive compared to built-in ints.
- Use direct initialization (T a(b);) rather than copy initialization (T a = b;). The latter syntax may create a temporary object, but the former does not.

- Dynamic memory allocation and deallocation can be a bottleneck. Consider writing class-specific operator new() and operator delete() functions, optimized for objects of a specific size. It may be possible to recycle blocks of memory instead of releasing them back to the heap when an object goes out of scope.
- The Standard string class is not a lightweight component. Because it has a lot of functionality, it comes with a certain amount of overhead (and because Standard Library container classes throw C++ strings, not C-style string literals, this overhead may be included in a program inadvertently). In many applications, strings are created, stored, and referenced, but never changed. As an extension, or as a programmer-directed optimization, it might be useful to create a lighter-weight unchangeable-string class.
- Reference counting is widely used as an optimization technique. In a single-threaded application, it can prevent making unnecessary copies of objects. But in multi-

threaded situations, the overhead of locking the shared data representation may add unnecessary overheads.